POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Solving Systems of Linear Equalities in Modular Arithmetic with Applications to Model Counting in Constraint Programming

MAHSHID MOHAMMADALITAJRISHI
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
génie informatique

Août 2019

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Solving Systems of Linear Equalities in Modular Arithmetic with Applications to Model Counting in Constraint Programming

présenté par **Mahshid MOHAMMADALITAJRISHI**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Daniel ALOISE, président
Gilles PESANT, membre et directeur de recherche
Louis-Martin ROUSSEAU, membre

DEDICATION

I dedicate this thesis to my beloved husband, Hamidreza, who has been a constant source of support and encouragement during the challenges of graduate school and life.

ACKNOWLEDGEMENTS

I would like to thank and express gratitude to my supervisor Gilles Pesant for welcoming me to his research laboratory. Gilles has dedicated a considerable amount of time and energy to the supervision of my research and his guidance, patience, and support throughout this research and the writing of this dissertation was invaluable. I look forward to our future collaboration and cooperation.

Very special thanks are due to my parents for their everlasting love and support. I am thankful for having you in my life. I would also like to thank my sister, Marjan, whose good examples have taught me to work hard for the things that I aspire to achieve.

RÉSUMÉ

Le comptage et l'échantillonnage de modèles sont deux problèmes fondamentaux en intelligence artificielle. La théorie de ces problèmes remonte aux années 1980. Il existe différents problèmes dans divers domaines, tels que l'apprentissage automatique, la planification, les statistiques, etc., dont on sait qu'ils sont difficiles à calculer. Même trouver une solution unique peut être une lutte pour de tels problèmes; compter le nombre de solutions est beaucoup plus difficile. Ainsi, le comptage approximatif des modèles pourrait être utile pour les résoudre.

L'ideé de ce travail vient des travaux précédents qui sont davantage axés sur les variables binaires. Ils utilisent des techniques basées sur le hachage en générant des contraintes XOR de manière aléatoire pour partitionner l'espace des solutions en petites cellules, puis en utilisant un solveur SAT pour compter à l'intérieur d'une cellule aléatoire. Les solveurs SAT sont utilisés pour les domaines binaires, mais nous proposons ici d'utiliser des solveurs CP pour les domaines non binaires.

Le but de cette recherche est de présenter un algorithme permettant de compter approximativement le nombre de solutions d'un modèle de CP. Dans la première étape, nous commençons à diviser l'espace des solutions en p petites cellules à chaque contrainte de mod p ajoutée conformément à l'arithmétique modulaire p. Ensuite, en utilisant l'algorithme d'élimination de Gauss-Jordan, nous essayons de simplifier le système de contraintes linéaires générées aléatoirement. De plus, nous introduisons un algorithme qui, en créant un graphe, filtre les domaines des variables dans une petite cellule aléatoire. Après avoir compté le nombre de solutions dans une petite cellule, nous estimons le nombre de solutions en multipliant le nombre de solutions dans une cellule par le nombre de cellules.

ABSTRACT

Model counting and sampling are two fundamental problems in artificial intelligence. The theory of these problems goes back to the 1980s. There are different problems in various areas like machine learning, planning, statistics and so on which are known to be computationally hard. Even finding a single solution can be a struggle for such problems; counting the number of solutions is much harder. Thus, approximate model counting could be useful to solve them. The idea of this work comes from previous works which are focused more on binary variables. They use hashing-based techniques by generating random XOR constraints to partition the solution space into small cells and then use a SAT solver to count inside a random cell. SAT solvers are used for binary domains but we propose here to use CP solvers for non-binary domains.

The goal of this research is to present an algorithm for approximately counting the number of solutions of a CP model. In the first step, we start to divide the solution space into p small cells at each added mod p constraint according to modular arithmetic p. Then by using the Gauss-Jordan elimination algorithm we try to simplify the system of randomly generated linear constraints. Moreover we introduce an algorithm that by creating a graph incrementally filters variable domains in one random small cell. After counting the number of solutions in one small cell we estimate the number of solutions by multiplying the number of solutions in one cell by the number of cells.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

Counting perfect matchings in bipartite graphs and counting satisfying assignments to Boolean formulae in conjunctive normal form are examples of $\#P$-complete problems which are as difficult as NP-complete problems. [2]

Model counting and sampling are two essential problems in artificial intelligence. Model counting is to count the number of solutions and sampling is to sample randomly in the solution space. Both problems have numerous applications [3–7] such as machine learning, planning, DNA profiling, statistics, and probabilistic inference.

One of the critical applications of model counting and sampling is probabilistic inference [8]. The rain or sprinkler could cause the grass to be wet is one of the classical examples in probabilistic inference. If we consider wet grass as an event, the probability of the grass being wet is obtained by counting satisfying assignments where the grass is wet and dividing it by the total number of satisfying assignments:

$$\Pr(\text{Grass is wet}) = \frac{\#\text{SAT assignments with WetGrass =True}}{\#\text{of SAT assignments}}$$

Now we explain how we can translate probabilistic inference to model counting. A *model* is an assignment to variables for which a formula evaluates to true. The problem of counting the number of models of the formula is called model counting. We consider Boolean formula $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_1) \wedge (\neg x_1 \vee x_3)$ as a simple example. x_1 to x_3 are binary variables and if we assign True to x_1, False to x_2 and True to x_3 the given propositional formula evaluates to True. If we assign False to all the variables again the given model evaluates to True as well. So we want to count all the satisfying assignments for a given model.

Also, we know that counting the possible solutions is rarely practical since the number of possible solutions is exponential in the number of variables, so exact probabilistic inference is computationally intractable. The efficient way to handle these problems in practice is to use approximations. In general the problem of counting satisfying assignments of propositional languages is $\#P$-complete.

In this research our attempt is to present an algorithm that by using the hashing-based technique approximately counts the number of solutions of a CP model.

Figure 1.1 indicates each "hashing constraint" for $p = 5$, randomly splitting the solution

space in subregions containing the same expected number of solutions. At each step after dividing the solution space to subregions, we can pick one of them and continue the process (see Figure 1.2). Once a cell is of the "right" size, we count its solutions and extrapolate for the whole problem.

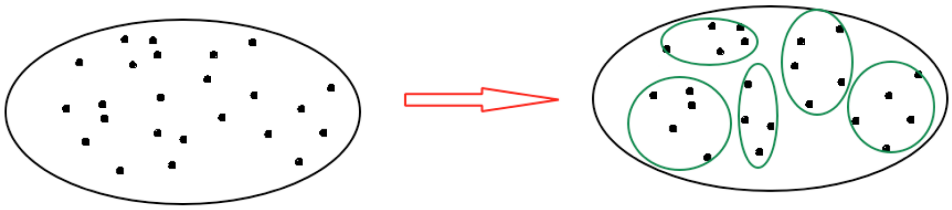

Figure 1.1 Add mod p=5 constraint

.

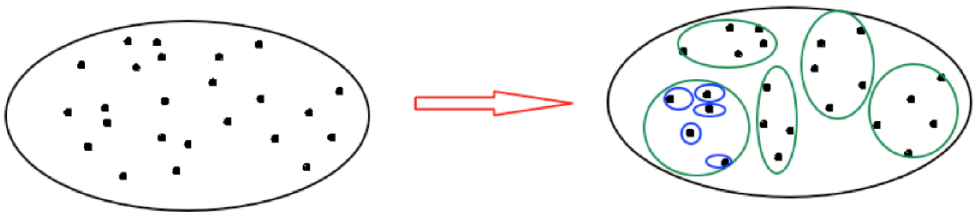

Figure 1.2 Add another mod p=5 constraint to the first constraint

.

1.1 Basics of Constraint Programming

Combinatorial optimization consists of finding an optimal object from a finite set of objects. In this type of problem the goal is to find the best solution. There are different computational methods to solve them. Among them, Constraint Programming formulates the problem as solving increasingly tighter Constraint Satisfaction Problems (CSP).

A CSP is typically expressed as a triple $\mathcal{P} =: (\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X} =: \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, D is a finite set of values such that x_i for each $i = 1, 2, \ldots, n$ can take a value from its domain $D_i \subseteq D$ and a finite set of constraints $\mathcal{C} =: \{C_1, C_2, \ldots, C_m\}$ with arity $1 \leq k \leq n$ defined on subset of $\mathcal{X}$. More formally, if $C \in \mathcal{C}$ is defined on a subset of variables $\{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \subseteq \mathcal{X}$ then $C \subseteq D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k}$.

CP offers many different constraints (linear and nonlinear constraints) [9]. Each type of con-

straint has its dedicated inference algorithm. CP uses the constraints to eliminate infeasible solution areas by removing some unsupported values from the domain of variables. Reduction of the search space reduces the computation time. This is valuable because sometimes the computation time grows exponentially with the size of the instance.

1.1.1 Inference

CP can be viewed as a network, where variables are like nodes and constraints are like edges (hyper-edges). Variables are labeled with a subset of their domain. The algorithm according to each constraint attempts to modify the labels which cannot be part of the solutions. CP model has a finite domain. So, the algorithm must terminate.

Domain consistency [10]. A constraint C on the variables $x_1, \ldots, x_k$ is domain consistent if for every $i \in \{1, \ldots, k\}$ and every $v \in D_i$

$$\forall j \in \{1, \ldots, k\} \backslash \{i\} \; \exists v_j \in D_j \;\text{ such that }\; c(v_1, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_k) \;\text{ is satisfied.}$$

Bound consistency. A constraint C on the variables $x_1, \ldots, x_k$ with respective domains $D(x_1), \ldots, D(x_k)$ is bound consistent if for every x_i, there exists a real number r_j, $j \in \{1, \ldots, k\} \backslash \{i\}$ belongs to $[D_{x_j}^{\min}, D_{x_j}^{\max}]$ such that $c(r_1, \ldots, r_{i-1}, D_{x_i}^{\min}, r_{i+1}, \ldots, r_k)$ is satisfied; and similarly for $D_{x_i}^{\max}$.

The overall process is called *Constraint Propagation.* During the constraint propagation, either all the labels are fixed (the problem is solved) or an empty domain is obtained (it has no solution) or there are still several values in some of the domains.

1.1.2 Modelling

In CP, one must define a formal mathematical problem to solve. Now, we present some of the most useful constraints to formulate problems [9].

Linear constraints. Consider a vector of integer finite-domain variables $\mathbf{X} = \langle X_1, X_2, \ldots, X_n \rangle$ and a vector of coefficients like, $\mathbf{c} = (c_1, c_2, \ldots, c_n)$. Write

$$l \leq \mathbf{cX} \leq u$$

where l and u are integers. We can achieve bound consistency in time linear in n. A domain consistency algorithm is explained in detail in the next section.

Table constraints. Consider a vector of integer finite-domain variables $\mathbf{X} = \langle X_1, X_2, \ldots, X_n \rangle$

$$\text{TABLE}(\mathbf{X}, \mathcal{T})$$

with set $\mathcal{T}$ of admissible n-tuples for X. The domain consistency algorithm runs in polynomial time [11].

Alldifferent constraints. Consider variables $x_1, x_2, \ldots, x_n$ with respective finite domains $D(x_1), D(x_2), \ldots, D(x_n)$.

$$\text{ALLDIFFERENT}(x_1, x_2, \ldots, x_n) = \{(d_1, \ldots, d_n) | d_i \in D(x_i), d_i \neq d_j, \forall i \neq j\}$$

We can achieve bound consistency on the alldifferent constraint in $O(n)$ plus the time needed to sort the bounds of the domains [12]. A domain consistency algorithm running in $O(n^{2.5})$ was presented by [13].

In the following example, we present constraint programming modeling for a small instance.

Example 1.1.1. *There are 10 variables $x_1 \ldots x_5, y_1 \ldots y_5$ and the domain of each variable is $[1, \ldots, 5]$.*

$$x_0 > x_1 + x_4$$

$$x_3 \leq x_0 \times x_2$$

$$\textit{Alldifferent}\{y_i,\ \forall i \in \{1, \ldots, 5\}\}$$

The number of solutions is 29400.

1.1.3 A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints

In this part, we review a dynamic programming structure to represent knapsack constraints [14]. This work is relevant for our handling of mod p hashing functions in CP. By this approach, we can achieve domain consistency, to determine infeasibility before all variables are set, to generate all solutions quickly. Additionally, we can provide incrementality by updating the structure after domain reduction.

A knapsack constraint is a linear constraint which is defined: $L \leq ax \leq U$ where L and U are scalars, $x = [x_1, x_2, \ldots, x_n]$ is an n-vector of variables, and $a = [a_1, a_2, \ldots, a_n]$ is an n-vector of *non-negative* integer coefficients. Each variable x_i is to take one value from a

finite domain D_i. First, we assume, for all i, $D_i = \{0, 1\}$, handling more general domains is a simple generalization of the notation.

Define a function $f(i, b)$ equal to 1 if the variables $1, \ldots, i$ can be set to values in their domains to exactly fill a knapsack of size b, and 0 otherwise ($0 \leq i \leq n$ and $0 \leq b \leq U$). We define;

$$f(0, 0) = 1$$

$$f(i, b) = \max\{f(i-1, b), f(i-1, b-a_i)\}$$

We can create a graph representing recursive function f then filtering it to achieve a reduced graph; this process is called *Knapsack-domain-consistency*. For the knapsack $10 \leq 2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 12$ the graph is shown in Figure 1.3;

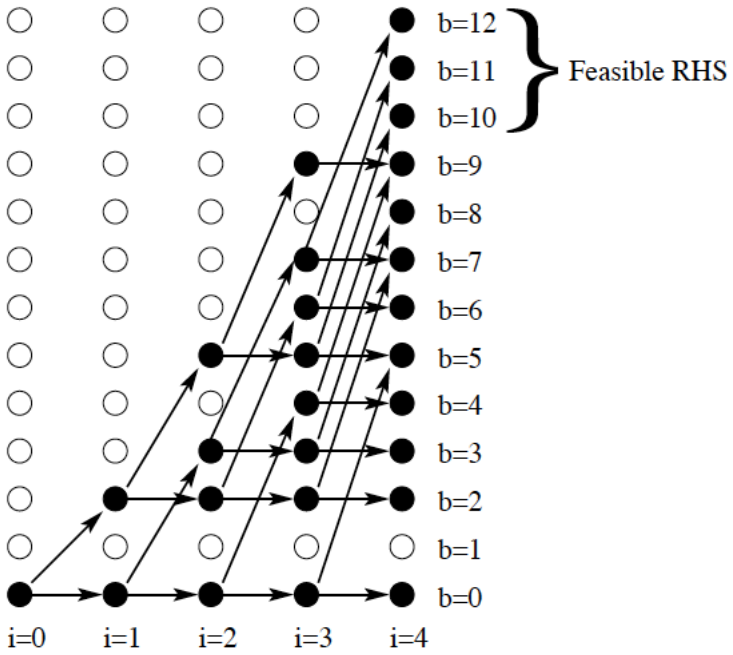

Figure 1.3 Knapsack Graph

In this graph, rows correspond to b values, while columns represent i and black nodes to f values equal to 1. As well as, the edges go from $(i-1, b)$ to (i, b) or from $(i-1, b)$ to $(i, b+a)$ between nodes with value 1.

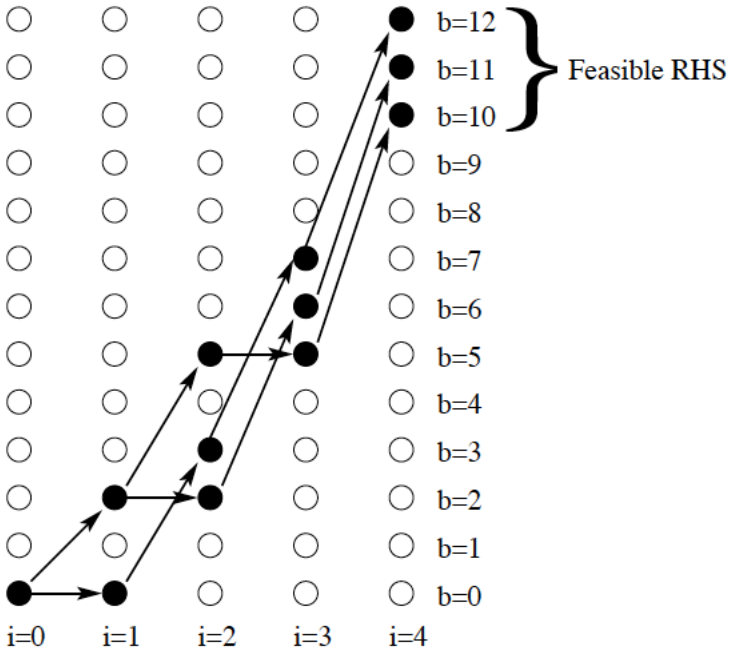

Figure 1.4 Reduced Knapsack Graph

The reduced graph is created by defining $g(i, b)$ as a recursive function in which i ranges from 0 to n, and b ranges from 0 to U. If the variables $i + 1, \ldots, n$ can fill a knapsack of size L to U, function $g(i, b)$ equals 1. Moreover, nodes (n, b) for b between L and U are called goal nodes.

Figure 1.4 shows the reduced graph. As we can see value 0 is filtered out from the domain of variable x_4. The domains after filtering are $\{0, 1\}, \{0, 1\}, \{0, 1\}, \{1\}$ respectively.

The time complexity of the algorithm is $O(nU^2)$.

This process may stop with indeterminate variables (whose domain still contains several values), hence the solution process requires search.

1.1.4 Search

We briefly describe how CP organizes a search. CP organizes tree search by branching on variables in order to resolve that indeterminacy. Backtracking search is performed as a depth-first search tree which is called *branching strategy*. We know each constraint has its internal data and algorithm. So, computation can be considerable. Also, depth-first search tries to detect a failed subtree as soon as possible to reduce computation. It means it omits some nodes. Sometimes the domain is large. So, *domain splitting* partitions the domain into two

or more branches. However, most of the time *branching* is binary. So, we fix a variable to a value in its domain in the left branch and remove that value from the domain in the right branch. *Fail-First Principle* is a subject of variable selection which recommends the failure branch comes sooner. One instance of the fail-first principle is *smallest-domain-first* heuristic, which selects the variable with the smallest-cardinality domain. *Value-selection heuristics* means the way to decide which value to try first, given a selected variable.

1.2 Research Objectives

A previously proposed approach for SAT models, inspired by universal hashing, adds randomly-generated XOR constraints to partition the solution space until each cell becomes tractable. Prior work on model counting focused on binary problems. Now, this is our attempt to extend it to finite domains for CSPs by considering randomly-generated linear constraints in modular arithmetic. Is it possible to use the Universal Hashing approach for CSPs? We investigate the opportunities to perform domain filtering and solution counting for "universal hashing" constraints in CP, using such techniques as Gaussian Elimination and Dynamic Programming.

1.3 Thesis Outline

The rest of this dissertation is organized as follow. Chapter 2 presents the relevant works in the areas of model counting and sampling. Chapter 3 introduces the modular p constraints generalized to non-binary domains and presents incremental filtering algorithms for CP. Chapter 4 discusses the experimental results before we finally conclude.

CHAPTER 2 LITERATURE REVIEW

In this chapter, we present related works to constrained sampling and counting. Constrained counting is the problem of counting the number of solutions of a propositional formula which is known as #SAT. In many applications of constrained counting, such as in probabilistic reasoning, approximate counts are sufficient. Other times, tractability forces one to work with approximate counting. Recently, by combining universal hashing as a solution-space reduction technique and SAT solvers as an oracle, they achieve scalable algorithms.

SAT Solving. Boolean Satisfiability (SAT) solving is one of the central problems in computer science. *Conflict-Driven Clause-Learning* (CDCL) solvers combine a backtracking search with a rich set of useful heuristics. These days, modern SAT solves millions of variables in a reasonable time. Propositional sampling and counting are extensions of the SAT problem.

SMT Solving. The Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas in combination with background theories. Recently in [15] the authors can directly leverage the power of sophisticated SMT solvers. In this approach the authors used *CryptoMiniSAT*, an SMT solver which combines SAT solving with XOR constraints, to achieve an effective solver for a combination of propositional CNF and XOR constraints.

2.1 Exact Algorithms

The earliest approach for #SAT was based on exactly counting the number of solutions. But exact counters can only solve small to medium size problems. Later to solve the bigger problems approximate algorithms based on exact algorithms have been introduced. DPLL is one of the exact counters on which most current SAT solvers are based. Now we briefly describe the DPLL algorithm.

2.1.1 #DPLL Algorithm

Consider formula $\{(w \bigvee x), (y \bigvee z)\}$. The following figure is a decision tree for the formula. In Fig 2.1, after ":" the sub-formula which is solved in the subtree is shown. The left branches correspond to value 0 or false and the right side corresponds to value 1 or true.

DPLL is an algorithm for SAT solving [16]. A *decision tree* over binary variables is a binary tree in which all the nodes are labeled with variables and edges with values 0 or 1. The tree over $x_1, \ldots, x_n$ represents possible assignments over a CNF formula $\phi(x_1, \ldots, x_n)$. The

#DPLL algorithm counts the probability of the set of satisfying assignments under the uniform distribution. Fig 2.2 indicates the $\#DPLL$ algorithm that returns a probability of the set of satisfying assignments under the uniform distribution.

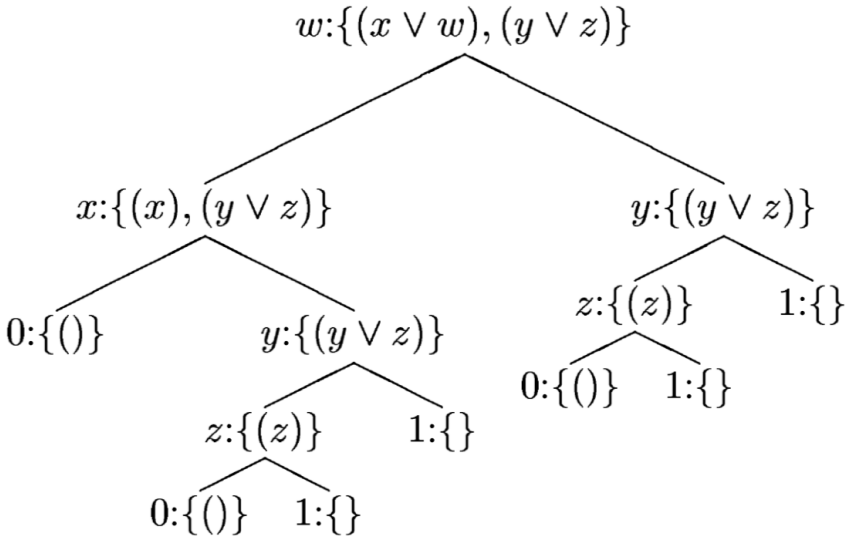

Figure 2.1 A decision tree for the given formula

.

#DPLL(ϕ)
Returns the probability of ϕ
if ϕ has no clauses, return 1
else-if ϕ has an empty clause, return 0
else
Choose a variable x that appears in ϕ
return #DPLL($\phi|_{x=0}$) $\times \frac{1}{2}$ + #DPLL($\phi|_{x=1}$) $\times \frac{1}{2}$

Figure 2.2 DPLL algorithm for computing $\#SAT$

.

2.2 Approximate Algorithms

To solve the bigger instances exact algorithms do not work, so we briefly introduce some approximate algorithms.

2.2.1 ApproxCount Algorithm

In [17], authors introduce an approximate algorithm ApproxCount to count approximately a number of satisfying assignments or models of formula in propositional logic. The ApproxCount algorithm is based on SampleSat [18] whose run time and accuracy are based on the number of samples the algorithm draws in each iteration. It means that we can have a result faster and less accurate by reducing the sample size. ApproxCount works incrementally and in each step sets one variable. The idea of the algorithm is to first draw K samples from the solution space of F such that each sample must be a satisfying truth assignment. Consider variable x_1 in F and $\#(x_1 = True)$ the number of samples in which x_1 is assigned to true. And also for False similarly. If $\mathcal{M}(F)$ denotes the number of unique satisfying assignments of formula F Then

$$\frac{\mathcal{M}(F \wedge x_1)}{\mathcal{M}(F)} \approx \frac{\#(x_1 = \text{True})}{K}, \quad \text{and} \quad \frac{\mathcal{M}(F \wedge \neg x_1)}{\mathcal{M}(F)} \approx \frac{\#(x_1 = \text{False})}{K}.$$

For satisfiability of the algorithm, in each step pick $\#(x_1 = True) \geq \#(x_1 = False)$. So the above formula is equivalent to

$$\mathcal{M}(F) \approx \frac{K}{\#(x_1 = \text{True})} \cdot \mathcal{M}(F \wedge x_1).$$

$M_{x_1} = \frac{\mathcal{M}(F)}{\mathcal{M}(F \wedge x_1)}$ is called multiplier of variable x_1. In general we can calculate:

$$\mathcal{M}(F) = M_{x_1}.M_{x_2}. \ \ldots \ .M_{x_n}$$

The computation time for the ApproxCount algorithm depends on a number of variables n, the number of samples drawn in each iteration K, and the time needed to draw a sample c.

Now, we introduce Hashing-based approaches that rely on *universal hashing*, *satisfiability modulo theories* (SMT) *solving*, and *satisfiability* (SAT) *solving*.

2.2.2 Universal Hashing

Universal hashing [19] is an algorithmic technique which randomly chooses a hash function from a family of functions with a specific mathematical property. It also guarantees a low expected number of collisions. Hash functions attempt to map any input from a large domain to a smaller range of output ($h : \{0,1\}^n \rightarrow \{0,1\}^m$ where $n > m$).

A hash family $\mathcal{H}$ is k-universal for $\mathcal{H} = \{h : U \rightarrow [m]\}$ where U is a universe, and $\forall x_i \in U$ we can write:

$$\Pr_{h \sim \mathcal{H}}(\exists i, j \;\; s.t. \;\; i \neq j \; and \; h_(x_i) = h(x_j)) \leq \frac{1}{n^{k-1}}$$

Hashing based techniques are used for those problems which are known to be computationally hard so using constrained sampling and counting can solve them approximately. By reducing the solution space and using the algorithm we can computationally count the number of solutions for different problems.

In [19] the authors use the universal hashing technique to map the large domain to a smaller one. They employ XOR constraints to partition the solution space into equally sized "small" cells. Then they count the number of solutions in one random cell and by multiplying by the number of cells approximately count the number of solutions. To partition the solution space m XOR constraints are generated. In each constraint, each binary variable is picked with probability $\frac{1}{2}$ and each constraint is equaled to 1 or 0 with probability $\frac{1}{2}$. E.g.: $X_1 \bigoplus X_3 \bigoplus X_5 \bigoplus \ldots \bigoplus X_{n-1} = 1$. Now, there are some questions which we want to talk about in the rest. For example: How can we solve each cell? How many cells do we need to solve?

The size of the cells is one of the crucial factors. If the size of the cells is too large, the enumeration is difficult, and also, if the size of the cell is too small, the ratio of variance to mean is very high. Therefore, a threshold $pivot = 5(1 + \frac{1}{\epsilon^2})$ (with $0 < \epsilon \leq 1$ a tolerance) is calculated.

Independent Support Hashing-based techniques rely upon the ability to find solutions for combinatorial solvers such as combination of CNF and XOR clauses. XOR clauses include typically $\frac{n}{2}$ variables that are high density clauses which affect runtime. Thus by restricting the hash functions to only the independent support $\mathcal{I}$ [1] we can improve the runtime performance [20]. In [21], an algorithm that gives the minimal independent supports is presented.

[1] In every satisfying assignment, the truth values of variables in $\mathcal{I}$ uniquely determine the truth value of every variable in $X \backslash \mathcal{I}$. The set $\mathcal{I}$ is called independent support of F, and $D = X \backslash \mathcal{I}$ is dependent support.

2.2.3 ApproxMC Algorithm

One of the approximate constrained counters is ApproxMC [15]. ApproxMC can handle tens of thousands of variables. This algorithm takes Boolean CNF formula F. ApproxMC uses 3-wise independent linear hashing functions from the $H_{xor}(n, m, 3)$ family that n and m are positive integers to randomly partition the set into "small" cells. Then, the algorithm chooses a random cell and checks that it is non-empty and has no more than *pivot* elements. If a random cell is not small (more than *pivot*), the algorithm must continue to randomly partition the solution space by choosing a random hash function from the family $H_{xor}(n, m+1, 3)$. This process will continue until the algorithm does not find non-empty and small cells or the number of cells exceeds $\frac{2^{n+1}}{pivot}$. In that case, the algorithm returns $\perp$.

Figure 2.3 illustrates this issue.

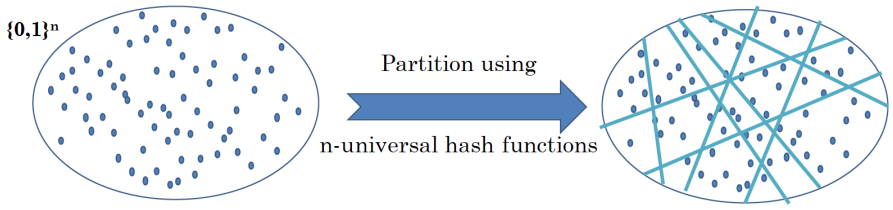

Figure 2.3 SAT Solver: randomly-generated XOR constraints reproduced from [1]

By first choosing a random cell, we can achieve an approximation of the size of R_F (solution space) and also measure the size of randomly chosen cells.

ApproxMC uses algorithm *ApproxMCCore* to determine the size of a "small" cell. If every propositional formula called F, ApproxMCCore takes CNF formula F and threshold pivot as inputs and returns an ϵ-approximate estimate of the model count of F. ApproxMCCore uses *BoundedSAT* to count the model. BoundedSAT takes a proposition formula F' that is the conjunction of a CNF formula, xor constraints, and a threshold $v \geq 0$ as inputs and returns a set S of models of F'

ApproxMC stores all non-$\perp$ estimates of the model count returned by ApproxMCCore in the list C. Finally, the final estimate which is the median of all the estimates stored in C is returned by ApproxMC.

2.2.4 UniGen Algorithm

UniGen is a new algorithm which falls into the category of hashing-based almost-uniform generators [20]. UniGen gets a CNF formula F and uses a family of a 3-wise independent

hash function to partition the set randomly.

UniGen emanates from earlier hashing based algorithms XORSample′ [22], UniWit [15] and PAWS [23]. But the key difference is that the latter could not handle large scale problems.

Consider $H_{xor}(n, m, 3)$ introduces family of 3-wise independent hash functions. Let $X = \{x_1, x_2, \ldots, x_{|X|}\}$ be a set of variables of F. By randomly choosing $h \in H_{xor}(n, m, 3)$ and $\alpha \in \{0, 1\}^m$ and conjunctive constraint $\bigwedge_{i=1}^{m}(h(x_1, x_2, \ldots, x_{|X|})[i] \leftrightarrow a[i])$ as an xor-clause, the satisfiability of a CNF formula with xor-clauses grows with the number of variables per xor-clause. An Independent support $\mathcal{I}$ of F is generally smaller than X also we can define the value of variables in $X \backslash \mathcal{I}$ by the value of variables $\mathcal{I}$. So by partitioning randomly its projection on $\mathcal{I}$ we can randomly partition R_F. By accepting a subset S (set of sampling variables of F) as an additional input, and using an independent support of F as the value of S we can randomly choose $h \in H_{xor}(|S|, m, 3)$ and $\alpha \in \{0, 1\}^m$ and conjunctive constraint $F \wedge \bigwedge_{i=1}^{m}(h(x_1, x_2, \ldots, x_{|S|})[i] \leftrightarrow a[i])$ to partition the R_F. If $|S| \ll |X|$ the expected number of variables in each xor-clause is reduced. Therefore, we can scale to larger problem sizes. UniGen can handle around 0.5M variables.

According to the previous part, in UniGen, parameter m is essential too. An efficient m leads algorithm to partition the set R_F quickly. UniGen first computes two quantities, *pivot* and κ to determine high and low thresholds for the size of each cell. Then, the algorithm calls ApproxMC to estimate C which determines a range of candidate values for m.

We can show that unweighted algorithms for approximate counting and almost-uniform sampling can work for weighted algorithms. It uses a SAT solver and a black-box weight function $w(.)$. For example, WISH is one of those algorithms which are used for constrained weighted counting. Wish uses an optimization Oracle. However, the oracle is expensive in practice [24].

2.2.5 Approximate Probabilistic Inference via Word-Level Counting

In this part, we present the approximate model counter that uses *word*-level hashing functions and SMT solver [25]. In the previous section, we explained about propositional model counting on Boolean domains. However, now the values of variables come from finite, large domains. Data values come from the domains naturally encoded as fixed-width words. Algorithm SMTApproxMC is an efficient word-level approximate model counting which we use to answer inference queries over high-dimensional discrete domains. This algorithm uses the 2-universal hash function with SMT solver.

A *word* is an array of bits. The size of the array is called the *width* of the word. We consider here *fixed-width* words, whose width is constant. It is easy to see that a word of width k can

be used to represent elements of a set of size 2^k. When a word of width k is treated as a vector, we assume the component bits are 0 to $k-1$. For dividing a term t by p which p is a prime number and can only divided by itself and 1. we can use $t \mod p$ which because of prime number p the distinct words don't behave similarly. The goal of this algorithm is to find the set of *model* or *solution* of F (compute $|R_F|$).

Universal hash functions play an important role in this work. Let $\sup(F) = \{x_0, \ldots x_{n-1}\}$, where each x_i is a word of width k. The space of all assignments of words in $\sup(F)$ is $\{0,1\}^{n.k}$. Hash functions map elements of $\{0,1\}^{n.k}$ to p bins labeled $0, 1, \ldots p-1$, where $1 \leq p < 2^{n.k}$. Let $\mathbb{Z}_p$ denote $\{0, 1, \ldots p-1\}$ and let $\mathcal{H}$ denote a family of hash functions that map $\{0,1\}^{n.k}$ to $\mathbb{Z}_p$.

At first, the family of hash functions $\mathcal{H}_{xor}$ is used to partition the solution space by XOR-ing a random subset of variables. Let X denote the n-dimensional vector $(x_0, \ldots x_{n-1})$. Consider each hash function of the form $h(X) = (\sum_{j=0}^{n-1} a_j * x_j + b) \mod p$ and the a_j's and b are elements of $\mathbb{Z}_p$. Observe that every $h \in \mathcal{H}$ partitions $\{0,1\}^{n.k}$ into p cells. The expected number of elements per cell is $2^{n.k}/p$. Since $p < 2^{n.k}$, every cell has at least 1 element.

Suppose we want to partition $\{0,1\}^{n.k}$ into p^c $(c > 1)$ cells. First, we need to define hash functions that map elements in $\{0,1\}^{n.k}$ to a tuple in $(\mathbb{Z}_p)^c$. To achieve this, consider a c-tuple of hash functions $\mathcal{H} \times \ldots \times \mathcal{H}$ (Cartesian product is taken c times) where, each of them maps $\{0,1\}^{n.k}$ to $\mathbb{Z}_p$. We know that the size of each cell is essential. By increasing c by 1, the size of each cell is reduced by factor of p and it will difficult to satisfy the above requirement if p is large. In order to obtain a family of hash functions that map $\{0,1\}^{n.k}$ to $(\mathbb{Z}_{p_j})$, The authors split each word x_i into slices of width $k/2^j$ and use a similar technique to map $\{0,1\}^{n.k}$ to $\mathbb{Z}_p$. In general, we may wish to define a family of hash functions that maps $\{0,1\}^{n.k}$ to $\mathcal{D}$, where $\mathcal{D}$ is given by $(\mathbb{Z}_{p_0})^{c_0} \times (\mathbb{Z}_{p_1})^{c_1} \times \ldots \times (\mathbb{Z}_{p_{q-1}})^{c_{q-1}}$ and $\prod_{j=0}^{q-1} p_j^{c_j} < 2^{n.k}$. The case of when k is not a power of 2 is handled by splitting the words x_i into slices of size $\lceil k/2 \rceil, \lceil k/2^2 \rceil$ and so on. The family of hash functions can be presented as $\mathcal{H}_{SMT}(n, k, C)$ since it depends on n, k and the vector $C = (c_0, c_1, \ldots c_{q-1})$. By setting c_i to 0 for all $i \neq \lceil \log_2(k/2) \rceil$, and c_i to r for $i = \lceil \log_2(k/2) \rceil$ it reduces $\mathcal{H}_{SMT}$ to the family $\mathcal{H}_{xor}$ which maps $\{0,1\}^{n.k}$ to $\{0,1\}^r$.

Recently, by combining universal hashing and SAT solving, we can scale to industrial-size instances. Whereas in prior work we could not handle large-size real-world instances. Despite these efforts, there still exists a large gap between theoretical and practical results in this area.

CHAPTER 3 UNIVERSAL HASHING-BASED MODEL COUNTING IN CONSTRAINT PROGRAMMING

In [25] the algorithm SMTApproxMC is presented as a hashing-based model counting for non-binary domains using an SMT solver. In this chapter we try to extend it to non-binary domains for problems expressed as CSPs.

We introduce the modulo p constraints generalized to non-binary domains and describe the graph data structure used to provide a compact representation of its set of solutions. Then we adapt Gauss-Jordan Elimination for a system of such constraints and provide an incremental filtering algorithm both for our system and for individual constraints. In the rest of the chapter we discuss counting solutions of two constraints with shared variables, and at the end we describe how to use these constraints for approximate model counting.

3.1 Linear Equality Constraints in Modular Arithmetic

Modulo p constraints. Consider a vector of integer finite-domain variables $\mathbf{X} = \langle X_1, X_2, \ldots, X_n \rangle$, integer p, a vector of coefficients $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ with $c_i \in F_p$ and $u \in F_p$. We write

$$\mathbf{cX} \equiv u \mod p$$

Which chosen u restricts the combinations of values taken by X to the ones verifying the corresponding equation in modular arithmetic.

Example 3.1.1. *Let the domain for all the variables be* $D = \{1, 2, 3\}$ *and* $p = 5$.

$$2x_1 + 3x_2 + x_3 \equiv 4 \ mod\ 5$$

This constraint has 5 solutions; $(3, 2, 2)$, $(1, 2, 1)$, $(1, 3, 3)$, $(2, 1, 2)$, *and* $(2, 3, 1)$.

3.1.1 Filtering and Counting for a Single Linear Equality

In [14] Trick presents a dynamic programming structure for knapsack constraints. In the following we borrow this dynamic programming formulation to represent our constraints with modulo p.

The variables in constraints have a coefficient, and their domain can be different as well. We define a function $f(i, b)$ where i ranges from 0 to n (number of variables) and b ranges from

0 to $p-1$. Now we can define the function recursively as follows:

$$f(0,b) = \begin{cases} 1 & \text{if } b = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$f(i,b) = \sum_{d \in D} f(i-1, (b - c_i \times d) \bmod p) \; 1 \leq i \leq n$$

specifically, D is the domain of x_i.

We can visualize this as a network, f represents the number of paths from the source. Edges go between nodes when the value of f is positive. Moreover we define function $g(i,b)$ where i ranges from 0 to n (number of variables) and b ranges from 0 to $p-1$. This function is used in backward manner to eliminate the superfluous edges (see Algorithm 1).

$$g(n,b) = \begin{cases} 1 & \text{if } b = u \\ 0 & \text{otherwise} \end{cases}$$

$$g(i,b) = \sum_{d \in D} g(i+1, (b + c_{i+1} \times d) \bmod p) \; 0 \leq i < n$$

specifically, D is the domain of x_{i+1}.

Figures 3.1-3.3 illustrate the graphs for the following simple example:

$$2x_1 + 4x_2 + x_3 + 3x_4 \equiv 2 \; mod \; 5 \quad D = \{0,1,2,3,4\}$$

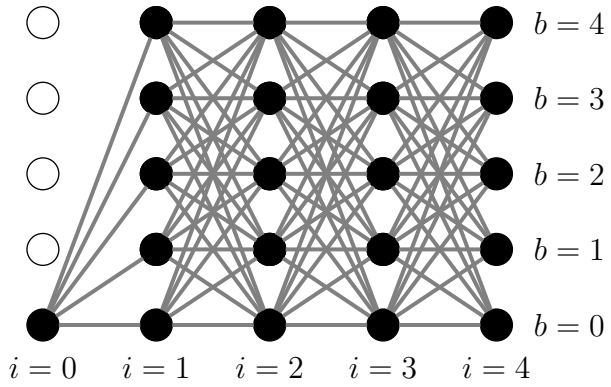

Figure 3.1 graphical representation of forward pass

In these graphs, rows correspond to b values, while columns represent i (variables). The

algorithm starts from $f(0,0)$ and proceeds forward to the last layer (Algorithm 1). In this specific example, the constraint equals 2; thus we can eliminate some paths which are not the solutions to our example.

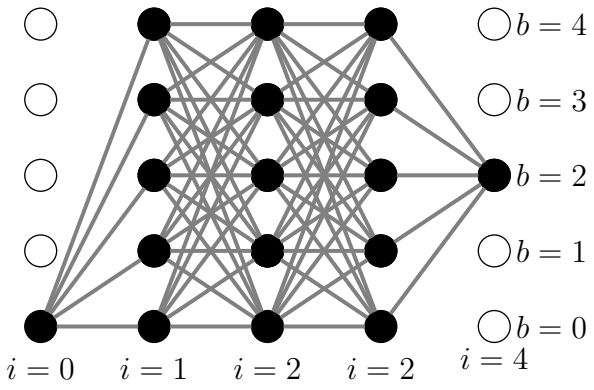

Figure 3.2 The graphical representation of backward pass

Figure 3.2 shows the reduced graph, where the algorithm starts from $g(4,2)$ and backward to eliminate the superfluous edges. According to the reduced graph, the number of solutions is equal to 125. At Figure 3.3, we show how the number of solutions can be computed. The number above each black node represents the number of paths received to that node i.e. the value of f. In the last layer, the total number of paths to the node $b = 2$ is 125. so the number of solutions equals 125.

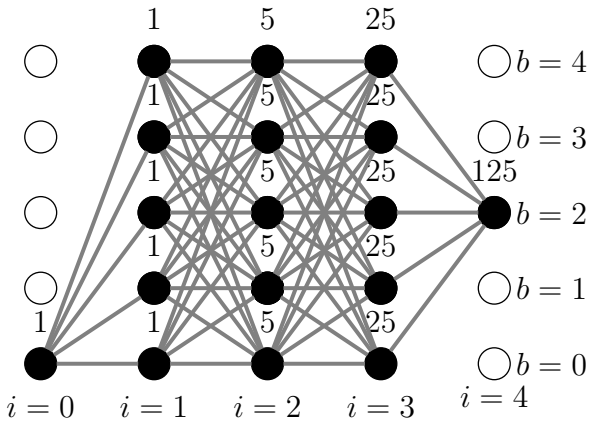

Figure 3.3 graphical representation for counting the number of solutions

The pseudocode for counting the number of solutions is presented in Algorithm 1. First,

we indicate the inputs and output of the algorithm. Here, c is a vector of coefficients, $D_0, D_1, \ldots, D_{n-1}$ is a set of domains, p is a prime number, u is the right-hand side of the constraint, and n is the number of variables. The output of the algorithm is a number of solutions of a constraint. The algorithm in a forward manner starts to create the graph according to the number of variables and p. Then in a backward manner, the algorithm starts from the last layer and according to the right-hand side, removes the superfluous edges. Since the number of paths represents the number of solutions, so $g(0, 0)$ gives the number of solutions. This algorithm also removes unsupported domain values to achieve domain consistency.

Algorithm 1: Domain filtering and solution counting algorithm

input : c is a vector of coefficients
D is the set of Domains: $D_0, D_1, \ldots, D_{n-1}$
p is the prime number
u is a right hand side
n is number of variables
output: The number of solutions

```
1  Forward
2  f(0,0) = 1;
3  f(0,b) = 0 forall b > 0;
4  for i ∈ 1 to n do
5  |  for b ∈ 0 to p − 1 do
6  |  |  f(i,b) = 0
7  |  |  for each d in D_{i−1} do
8  |  |  |  f(i,b) += f(i − 1, (b − c_{i−1} × d) mod p);
9  Backward
10 if f(n,u) > 0 then
11 |  g(n,b) = 0 for b ≠ u;
12 |  g(n,u) = 1;
13 |  for i ∈ n − 1 to 0 do
14 |  |  for b ∈ 0 to p − 1 do
15 |  |  |  if f(i,b) > 0 then
16 |  |  |  |  incomplete = true;
17 |  |  |  |  for each d in D_i do
18 |  |  |  |  |  if g(i + 1, (b + c_i × d) mod p) > 0 then
19 |  |  |  |  |  |  incomplete = false;
20 |  |  |  |  |  |  g(i,b) += g(i + 1, (b + c_i × d) mod p);
21 |  |  |  |  if (incomplete) then
22 |  |  |  |  |  f(i,b) = 0;
23 for i ∈ 0 to n − 1 do
24 |  for each d in D_i do
25 |  |  noSupport = true;
26 |  |  for b ∈ 0 to p − 1 do
27 |  |  |  if f(i,b) > 0 and f(i + 1, b + c_i × d) mod p) > 0 then
28 |  |  |  |  noSupport = false;
29 |  |  |  |  break;
30 |  |  if noSupport then
31 |  |  |  x[i].removeValue(d);
32 return g(0,0);
```

Whereas the original algorithm of Trick [14] is pseudo-polynomial since its time and space complexities are related to the magnitude of the right-hand side, here it is bounded by p which itself is defined relative to n and $|D|$. The time complexity of this algorithm is $\mathsf{O}(np|D|)$.

Consider the following example. The domain for all the variables is $D = \{1, 2, 3\}$ and $p = 7$.

$$6x_1 + 0x_2 + 1x_3 + 0x_4 + 6x_5 \equiv 0 \ mod \ 7$$

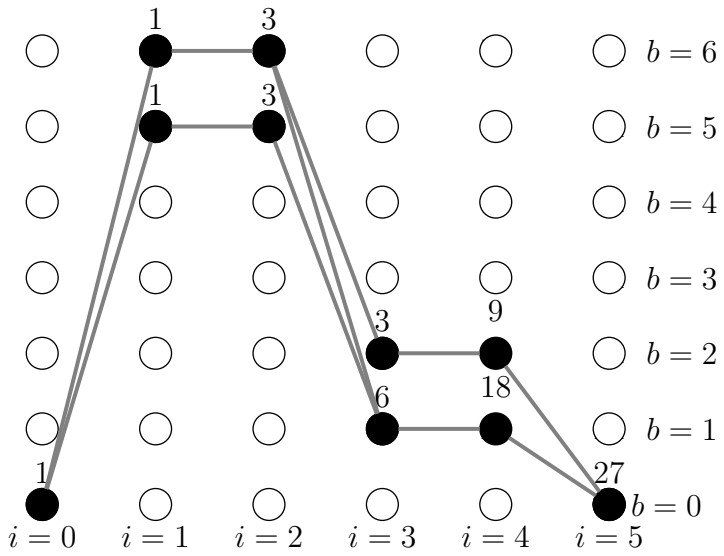

Figure 3.4 graphical representation of the set of solutions for this constraint upon completion of Algorithm 1

In first layer:

f(0,0) to f(1,6) on value 1

f(0,0) to f(1,5) on value 2

f(0,0) to f(1,4) on value 3

In second layer:

f(1,4) to f(2,4) on value 1,2,3

f(1,5) to f(2,5) on value 1,2,3

f(1,6) to f(2,6) on value 1,2,3

In third layer:

f(2,4) to f(3,5) on value 1

f(2,4) to f(3,6) on value 2

f(2,4) to f(3,0) on value 3

f(2,5) to f(3,6) on value 1

f(2,5) to f(3,0) on value 2

f(2,5) to f(3,1) on value 3

f(2,6) to f(3,0) on value 1

f(2,6) to f(3,1) on value 2

f(2,6) to f(3,2) on value 3

In furth layer:

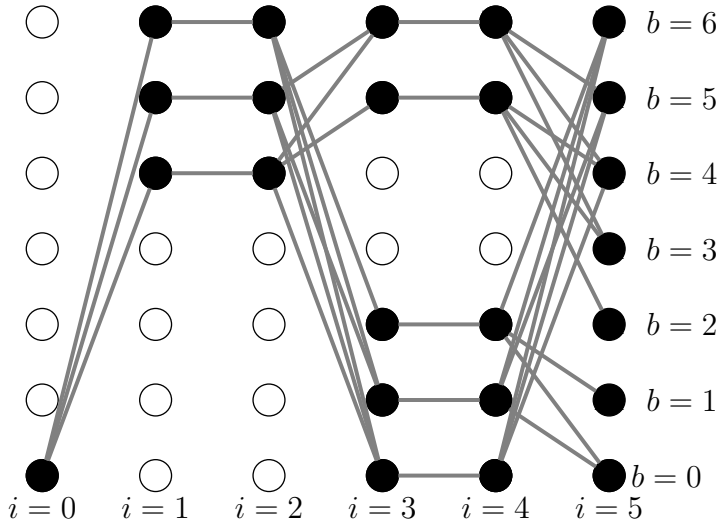

Figure 3.5 graphical representation of the computation of function f (forward pass of Algorithm 1)

f(3,0) to f(4,0) on value 1,2,3
f(3,1) to f(4,1) on value 1,2,3
f(3,2) to f(4,2) on value 1,2,3
f(3,5) to f(4,5) on value 1,2,3
f(3,6) to f(4,6) on value 1,2,3

In fifth layer:
f(4,0) to f(5,6) on value 1
f(4,0) to f(5,5) on value 2
f(4,0) to f(5,4) on value 3
f(4,1) to f(5,0) on value 1
f(4,1) to f(5,6) on value 2
f(4,1) to f(5,5) on value 3
f(4,2) to f(5,1) on value 1
f(4,2) to f(5,0) on value 2
f(4,2) to f(5,6) on value 3
f(4,5) to f(5,4) on value 1
f(4,5) to f(5,3) on value 2
f(4,5) to f(5,2) on value 3
f(4,6) to f(5,5) on value 1
f(4,6) to f(5,4) on value 2

f(4,6) to f(5,3) on value 3

Since in this example $b = 0$, in the last layer all the edges are removed except those connected to node $b = 0$. So value 3 is removed from the domain of variable 5. Since the latter edge has been removed, value 1 is removed from the domain of variable 3, and value 3 is removed from the domain of variable 1 as well. And the number of solutions is 27.

3.2 Gauss-Jordan Elimination for Systems of Such Constraints with Same Modulo

In this section, we introduce a Gauss-Jordan Elimination algorithm and explain how this algorithm works with modulo p. The advantage of using Gauss-Jordan Elimination algorithm in our work is that we can simplify the constraints system.

3.2.1 Gauss-Jordan Elimination with Modular Arithmetic

Gauss-Jordan Elimination is an algorithm to solve systems of linear equations over the rational or real numbers. The algorithm applies on a matrix of coefficients by the sequence of operations to modify the matrix until the main diagonal is filled with 1 and the triangle below and above the main diagonal is filled with 0. This algorithm is known as row reduction as well because the operations contain, swapping two rows, multiplying a row by a nonzero number and adding a multiple of one row to another row.

Here are the main steps of Gauss-Jordan Elimination;

$$\left[\begin{array}{cccc|c} \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# \end{array}\right] \text{ make the lower triangle 0 } \sim \left[\begin{array}{cccc|c} \# & \# & \# & \# & \# \\ 0 & \# & \# & \# & \# \\ 0 & 0 & \# & \# & \# \end{array}\right] \text{ make the main diagonal 1}$$

$$\sim \left[\begin{array}{cccc|c} 1 & \# & \# & \# & \# \\ 0 & 1 & \# & \# & \# \\ 0 & 0 & 1 & \# & \# \end{array}\right] \text{ make the upper triangle 0 } \sim \left[\begin{array}{cccc|c} 1 & 0 & 0 & \# & \# \\ 0 & 1 & 0 & \# & \# \\ 0 & 0 & 1 & \# & \# \end{array}\right]$$

3.2.2 Reducing the System of Equations through Gauss-Jordan Elimination

Now we use Gauss-Jordan Elimination with modular arithmetic operations to solve the constraints. The advantage of modular arithmetic is that we never perform division and all

coefficients remain in F_p. We know that Gauss-Jordan Elimination cannot be used on integer variables unless we are in modular arithmetic.

The number of variables and constraints respectively represent the number of columns and rows, so in general, the matrix is not square. Hence, the variables are divided into two parts, parametric and non-parametric.

Example 3.2.1. *In this example, there are two constraints with three variables in common with modulo $p = 5$. There are two rows and four columns (3 variables and one column for right-hand side).*

$$\begin{cases} 2x_1 + 3x_2 + 2x_3 \equiv 2 \ mod\ 5 \\ x_1 + x_2 + 4x_3 \equiv 3 \ mod\ 5 \end{cases}$$

$$\left[\begin{array}{ccc|c} 2 & 3 & 2 & 2 \\ 1 & 1 & 4 & 3 \end{array}\right] 3R_1 \quad \sim \quad \left[\begin{array}{ccc|c} 1 & 4 & 1 & 1 \\ 1 & 1 & 4 & 3 \end{array}\right] -R_1 + R_2$$

$$\sim \left[\begin{array}{ccc|c} 1 & 4 & 1 & 1 \\ 0 & 2 & 3 & 2 \end{array}\right] 3R_2 \quad \sim \quad \left[\begin{array}{ccc|c} 1 & 4 & 1 & 1 \\ 0 & 1 & 4 & 1 \end{array}\right] -4R_2 + R_1$$

$$\sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 4 & 1 \end{array}\right]$$

From the first equation, we have $x_1 = 2 \bmod 5$. And from the second equation we can write $x_2 + 4x_3 = 1$ or $x_2 = 1 + x_3 \bmod 5$. Thus the solutions are $(x_1, x_2, x_3) = (2, 1 + t \ mod\ 5, t)$ with $t \in F_5$.

Example 3.2.2. *Here is a slightly larger example with two parametric variables with modulo 5. Two constraints with four variables are presented here; All the coefficients are in F_5.*

$$\begin{cases} 2x_1 + 3x_2 + 2x_3 + x_4 \equiv 4 \ mod\ 5 \\ 4x_1 + 2x_2 + 3x_3 + x_4 \equiv 2 \ mod\ 5 \end{cases}$$

We present the two constraints in a matrix, then use Gauss-Jordan Elimination with modulo

5.

$$\left[\begin{array}{cccc|c} 2 & 3 & 2 & 1 & 4 \\ 4 & 2 & 3 & 1 & 2 \end{array}\right] 3R_1 \quad \sim \quad \left[\begin{array}{cccc|c} 1 & 4 & 1 & 3 & 2 \\ 4 & 2 & 3 & 1 & 2 \end{array}\right] -4R_1 + R_2$$

$$\sim \left[\begin{array}{cccc|c} 1 & 4 & 1 & 3 & 2 \\ 0 & 1 & 4 & 4 & 4 \end{array}\right] -4R_2 + R_1 \quad \sim \left[\begin{array}{cccc|c} 1 & 0 & 0 & 2 & 1 \\ 0 & 1 & 4 & 4 & 4 \end{array}\right]$$

After using Gauss-Jordan Elimination, we can reach a more straightforward format of constraints.

$$\begin{cases} x_1 + 2x_4 \equiv 1 \ mod\ 5 \\ x_2 + 4x_3 + 4x_4 \equiv 4 \ mod\ 5 \end{cases}$$

From the first equation, we have $x_1 = 1 + 3x_4 \bmod 5$, and from the second one we have $x_2 = 2 + x_3 + 2x_4 \bmod 5$.

3.2.3 Achieving Domain Consistency

We now describe our algorithm to achieve Domain(D) Consistency. The algorithm starts with the domain of parametric variables, for each value of the domain set the algorithm solves the constraints and computes possible solutions. If we start from the solved form after the application of Gauss-Jordan Elimination, all the coefficients of non-parametric variables except one are zero. So, the algorithm finds the solutions quickly. However, the possible values might not exist in the domain of non-parametric variables. So, the algorithm starts filtering. For each candidate solution, the algorithm checks whether the corresponding values appear in the domains of each non-parametric variable. If for all non-parametric variables values in both sets were equal, algorithm stores the possible result as a real result.

For Example 3.2.1, assume the respective domains are:

$$D = \begin{pmatrix} 0, 2, 3 \\ 0, 1, 4 \\ 2, 3, 4 \end{pmatrix}$$

The algorithm starts to solve the problem according to the given domain of parametric variables. Here it is $\{2, 3, 4\}$ for x_3.

The possible solutions according to the given domain for parametric variable are

$$\left\{ \ (2,3,2),(2,4,3),(2,0,4) \ \right\}$$

According to this set of solutions, it is clear x_1 can accept just value 2. For variable x_2, value 3 is in the candidate solution set, but it is not in the domain of variable x_2. Note also that value 1 in the domain of x_2 does not appear in any candidate solution. Therefore, the algorithm needs to filter. In the next step after the filtering the solutions for this example according to the domain are

$$\left\{ \ (2,4,3),(2,0,4) \ \right\}$$

Here is the Domain(D) after filtering;

$$D = \begin{pmatrix} 2 \\ 0,4 \\ 3,4 \end{pmatrix}$$

For the second example, we have two parametric variables (x_3 and x_4). The algorithm for each $v \in D(x_4)$ fixes the value, then continues like before (one parametric variable).

consider the respective domains are:

$$D = \begin{pmatrix} 1,3,4 \\ 0,3,4 \\ 1,2,3 \\ 1,3,4 \end{pmatrix}$$

According to the given domain for this specific example we have 3 different cases. Then, the algorithm continues as in the previous example. When the algorithm fixes variable x_4 to value 1 the constraints will change as follow;

$$\begin{cases} x_1 + 0x_3 \equiv 4 \ mod \ 5 \\ x_2 + 4x_3 \equiv 0 \ mod \ 5 \end{cases}$$

The Domains(D) after filtering are

$$D = \begin{pmatrix} 4 \\ 3 \\ 3 \\ 1 \end{pmatrix}$$

If $x_4 = 3$, we can write

$$\begin{cases} x_1 + 0x_3 \equiv 0 \ mod\ 5 \\ x_2 + 4x_3 \equiv 2 \ mod\ 5 \end{cases}$$

According to the first constraint, x_1 is 0, but value 0 is not in domain of variable x_1. So x_4 could not be 3.

If $x_4 = 4$, we can write

$$\begin{cases} x_1 + 0x_3 \equiv 3 \ mod\ 5 \\ x_2 + 4x_3 \equiv 3 \ mod\ 5 \end{cases}$$

The Domains(D) after filtering are

$$D = \begin{pmatrix} 3 \\ 0, 4 \\ 1, 2 \\ 4 \end{pmatrix}$$

The Domains(D) for example 3.2.2 are

$$D = \begin{pmatrix} 3, 4 \\ 0, 3, 4 \\ 1, 2, 3 \\ 1, 4 \end{pmatrix}$$

Value 1 is filtered out of the domain of x_1, and value 3 is filtered out of the domain of x_4.

The time complexity of the algorithm is in $\Theta(md^2)$ where the number of parametric variables $n' \leq n - m + 1$, m is the number of constraints, and d is the domain size. This algorithm is too expensive unless n' is small so we do not use it in our implementation. Note that this algorithm performs domain filtering for only one parametric variable that the time complexity is $\Theta(nd^2)$. More than one parametric variable requires more computational effort.

3.2.4 Dynamic Programming on Individual Constraints of Gauss-Jordan Elimination Solved Form

As we have explained in the previous section achieving domain consistency is costly when $n - m$ is large. So, we use dynamic programming on individual constraints as we saw in Section 3.1.1 which is less computationally costly than the previous one.

According to the previous Example 3.2.1, after using Gauss-Jordan Elimination, we can

rewrite two constraints;

$$\begin{cases} x_1 + 0x_3 \equiv 2 \ mod \ 5 \\ x_2 + 4x_3 \equiv 1 \ mod \ 5 \end{cases}$$

With domain;

$$D = \begin{pmatrix} 0, 2, 3 \\ 0, 1, 4 \\ 2, 3, 4 \end{pmatrix}$$

The following graphs represent the solutions for this specific example when the domains for all the variables are the same as before. This solved form allows us to have smaller layered graphs (on fewer variables) for individual linear equalities.

$$x_1 + 0x_3 \equiv 2 \ mod \ 5$$

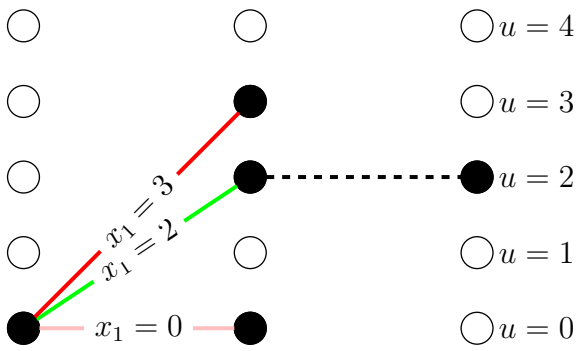

Figure 3.6 graphical representation for the first constraint of Example 3.2.1

The constraint has one solution, and values 0 and 3 are filtered out of the domain of x_1. The coefficient of parametric variable x_3 is zero, so there is no filtering (The black dashed line indicates the coefficient is zero).

$$x_2 + 4x_3 \equiv 1 \ mod \ 5$$

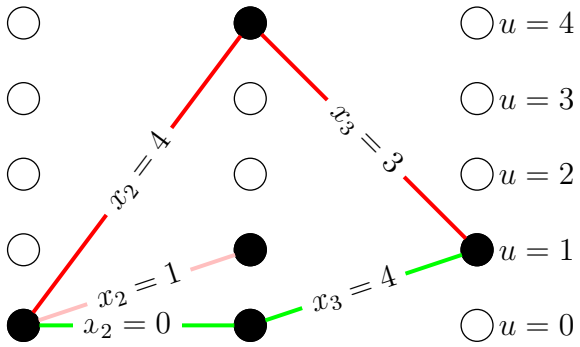

Figure 3.7 graphical representation for the second constraint of Example 3.2.1

The constraint has two solutions, and value 1 is filtered out of the domain of x_2. Furthermore, value 2 is filtered out of the domain of x_3

For bigger example 3.2.2 Gauss-Jordan Elimination is used to make the constraints simpler. Now we use the graphs to count the solutions and filter the domains.

$$\begin{cases} x_1 + 0x_3 + 2x_4 \equiv 1 \ mod\ 5 \\ x_2 + 4x_3 + 4x_4 \equiv 4 \ mod\ 5 \end{cases}$$

It is clear that the constraint with four variables has a dense graph, but Gauss-Jordan Elimination makes the graph more sparse. The following graphs represent the two constraints after using Gauss-Jordan Elimination, according to the given domain;

$$D = \begin{pmatrix} 1, 3, 4 \\ 0, 3, 4 \\ 1, 2, 3 \\ 1, 3, 4 \end{pmatrix}$$

For the first constraint;

$$x_1 + 0x_3 + 2x_4 \equiv 1 \ mod\ 5$$

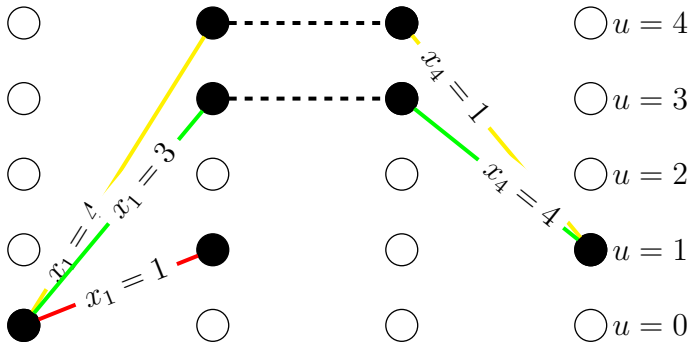

Figure 3.8 graphical representation for the first constraint of Example 3.2.2

The graph shows the constraint has two solutions, and value 1 is filtered out of the domain of x_1. The coefficient of parametric variable x_3 is zero, so there is no filtering, also value 3 is filtered out of the domain of x_4.

For the second constraint;

$$x_2 + 4x_3 + 4x_4 \equiv 4 \; mod \; 5$$

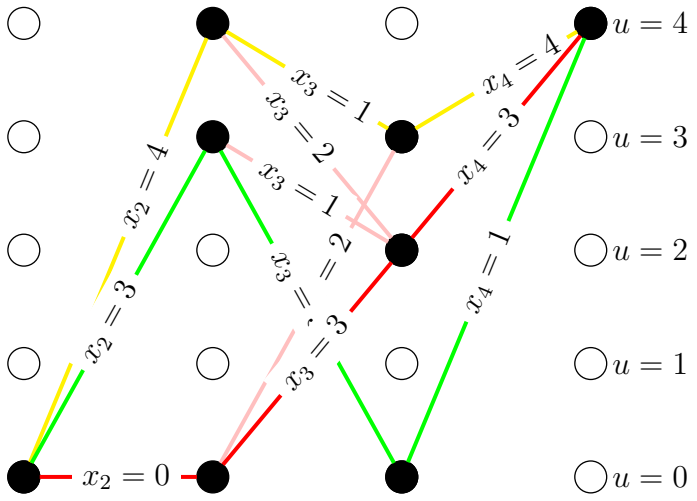

Figure 3.9 graphical representation for the second constraint of Example 3.2.2

The second constraint has six solutions.

3.3 Incremental Filtering Algorithm for Our Constraints

Now we briefly explain about the incrementality of the algorithm. As we mentioned before, we modify the creation of the graph by using doubly linked lists. Each node is thus linked to the previous and next layer nodes.

Consider i the index of the variable whose domain has been modified. The algorithm incrementally removes the disconnected paths. It means checking the right and left subpaths and if they need to be deleted, remove them.

Consider the following constraint:

$$A: \quad 2x_1 + 4x_2 + x_3 + 3x_4 \equiv 3 \; mod \; 7$$

And the domains of variables $D_{x_1} = \{1, 2, 3\}$ $D_{x_2} = \{2, 3\}$ $D_{x_3} = \{1, 2\}$ $D_{x_4} = \{2, 3, 1\}$

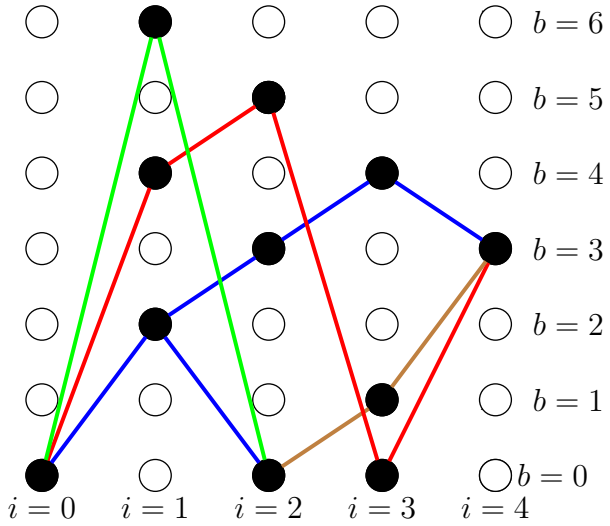

Figure 3.10 The graphical represents for incremental filtering algorithm of constraint A

The following graph shows once value 2 is removed from the domain of variable x_3. The dashed red line will be removed from the graph.

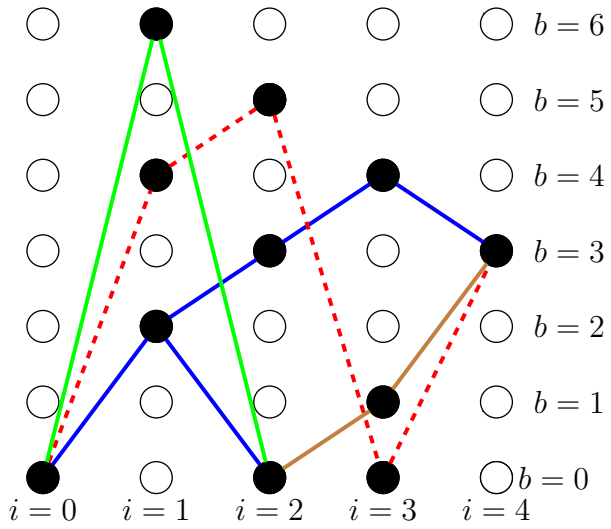

Figure 3.11 The graphical representation when a value is removed from a domain in incremental filtering (Algorithm 3)

By removing value 2 from domain of variable x_3, all unsupported values are removed from the domains of variables. The domains consistency after filtering is $D_{x_1} = \{1, 2, 3\}$, $D_{x_2} = \{2, 3\}$, $D_{x_3} = \{1, 2\}$, $D_{x_4} = \{1, 2, 3\}$.

To optimize the algorithm, we use a doubly linked list structure to reduce the time spent updating the graph. A doubly linked list is a linked list data structure that links to the previous nodes as well as to the next nodes.

Each node indicates by three elements, the link to the previous node, the link to the next node, and the value. Here we present the location of each node in a graph by (i, b). Furthermore, “Arcs-Out" presents the output arcs which are linked from the node to the next layer’s nodes, and “Arcs-In" presents all the input arcs which are linked from the previous layer’s nodes to the current node.

As you can see in Algorithm 2 lines 1-18 going forward the algorithm iterates over the domain of x_0 to create the Arcs-Out of the first layer. The Arcs-In of the first layer is empty, so we separate it. For the rest of the layers, for example in layer i, the algorithm checks whether the Arcs-Out node i is greater than zero or not. For the Arcs-In and Arcs-Out of nodes in layer $i + 1$ checks if Arcs-In and Arcs-Out node $i + 1$ is empty, adds the node. Else, makes next of new node as next of the previous node and makes the next of the previous node as a new node. Moreover the algorithm considers Arcs-Out in the last layer is empty.

In Algorithm 2 lines 19-36, in a backward manner the algorithm starts from the last layer

and according to the right-hand side, removes the arcs in the last layer then according to the coefficients and value in the domain set of each variable checks if Arcs-Out is positive meaning at least one path already exists and continues. Otherwise it removes the paths. In the end the algorithm checks the unsupported values at lines 37-45.

Algorithm 3 presents updating the graph from a given arc in layer i being removed. In this algorithm the input is i which is the index of the variable whose domain has been modified. The algorithm iterates over values in the domain of variable i and checks if the Arcs-Out at node (i, b) is not empty and then removes the left and right subpaths of the current node. Algorithm 5 shows how the right-side subpath is removed. Also, Algorithm 6 shows how the left-side subpath is removed. Both algorithms use a recursive manner to remove the subpaths.

Algorithm 2: Incremental domain filtering and solution counting algorithm

input : c is a vector of coefficients, D is the set of Domains: $D_0, D_1, \ldots, D_{n-1}$, p is the prime number, u is a right hand side, n is number of variables P-In is a object of Arcs-In, P-Out is a object of Arcs-Out, tab-Arcs-In is pointer to a pointer class Arcs-In, tab-Arcs-Out is pointer to a pointer class Arcs-Out

```
1  for i ∈ 0 to n do
2      for b ∈ 0 to p − 1 do
3          if tab-Arcs-Out(i, b) ≠ null then
4              for each v ∈ D do
5                  if tab-Arcs-In(i + 1, b + c_i × v mod p) == null then
6                      P-In = new Arcs-In(v, b, NULL);
7                  else
8                      P-In = newArcs-In(v, b, (i + 1, tab-Arcs-In(i + 1, b + c_i × v mod p));
9                      (P-In → getNext()) → setPrev(P-In );
10                 tab-Arcs-In(i + 1, b + c_i × v mod p).setValue(p-In);
11                 if
                    (i < n−1 and tab-Arcs-In(i+1, b+c_i×v mod p) ≠ NULL and tab-Arcs-Out(i+1, b+c_i×v mod) == NULL)
                    then
12                     for each v ∈ D do
13                         if Arcs-Out(i + 1, b + c_i × v mod p) == null then
14                             P-Out = new Arcs-In(v, (b + c_i × v mod p) + c_{i+1} × v mod p), NULL);
15                         else
16                             P-Out = new Arcs-Out(v, b+c_i × v mod p+c_{i+1} × v mod p), tab-Arcs-out(i+1.b+c_i × v mod p));
17                             (P-Out → getNext()) → setPrev(P-Out );
18                         tab-Arcs-Out(i + 1, b + c_i × v mod p).setValue(p-out);
19 for b ∈ 0 to p − 1 do
20     if b ≠ u then
21         for q ∈ Arcs-Out(n − 1, b) do
22             RemoveArc(q, Arcs-Out(n − 1, b)
23 for i ∈ n − 1 to 0 do
24     for b ∈ 0 to p − 1 do
25         if tab-Arcs-In(i, b) > 0 then
26             inComplete = True;
27             for each d in D_{i−1} do
28                 if tab-Arcs-Out(i, b) > 0 then
29                     inComplete = False;
30                     Break;
31             if (inComplete) then
32                 for q ∈ Arcs-In(i, b) do
33                     for p ∈ Arcs-Out(i − 1, q → original) do
34                         if (q == p) then
35                             RemoveArc(p, Arcs-Out(i − 1, q → original)
36                 tab-Arcs-In(i, b).setValue(NULL);
37 for i ∈ 0 to n − 1 do
38     for each d in D_i do
39         noSupport = true;
40         for b ∈ 0 to p − 1 do
41             if Arcs-Out(i, b) ≠ 0 and Arcs-In(i + 1, (b + c_i × d mod p)) ≠ 0 then
42                 noSupport = False;
43                 Break;
44         if noSupport then
45             x[i].removeValue(d);
```

Algorithm 3: updating the graph from a given arc

```
input : i: is the index of the variable whose domain has been modified
for each v ∈ deleted from D do
    for b ∈ 0 to p − 1 do
        for p ∈ Arcs-Out(i, b) do
            if p → getValue = v then
                RemoveLeftPath(b, i)
                RemoveArc(p, Arcs-Out(i, b))
        for q ∈ Arcs-In(i + 1, (b + c_i × v) mod p)) do
            if Arcs-In(i + 1, (b + c_i × v) mod p) ≠ null then
                if q → getValue = v then
                    RemoveRightPath((b + c_i × v) mod p, i + 1)
                    RemoveArc(q, Arcs-In(i + 1, (b + c_i × v) mod p))
for i ∈ 0 to n − 1 do
    for each d in D_i do
        noSupport = true;
        for b ∈ 0 to p − 1 do
            if Arcs-Out(i, b) ≠ 0 and Arcs-In(i + 1, (b + c_i × d mod p)) ≠ 0 then
                noSupport = False;
                Break;
        if noSupport then
            x[i].removeValue(d);
```

Algorithm 4: Remove Arc

```
input : p: Arc,
Arcs-list: the list of the arcs
if ((p → getPrev() ≠ NULL) && (p → getNext() ≠ NULL)) then
    (p → getPrev()) → setNext(p → getNext());
    (p → getNext()) → setPrev(p → getPrev());
else
    if ((p → getPrev() == NULL) and (p → getNext() ≠ NULL)) then
        (p → getNext()) → setPrev(NULL);
        Arcs-list.setValue(p → getNext());
    else
        if ((p → getPrev() ≠ NULL) and (p → getNext() == NULL)) then
            (p → getPrev()) → setNext(NULL);
        else
            Arcs-list.setValue(NULL);
```

Algorithm 5: Remove right path

```
input : i: is the index of the variable
        b: in 0 ... p − 1
1 if i < n then
2     if Arcs-In(i,b).size > 1 then
3         for p ∈ Arcs-Out(i, b) do
4             RemoveRightPath(p → destination, i + 1)
5             for q ∈ Arcs-In(i + 1, p → destination) do
6                 if (q == p) then
7                     RemoveArc(q, Arcs-In(i + 1, p → destination)
8             RemoveArc(p, Arcs-Out(i, b))
```

Algorithm 6: Remove left path

```
input : i: is the index of the variable
        b: in 0 ... p − 1
1 if i > 0 then
2     if Arcs-Out(i,b).size > 1 then
3         for p ∈ Arcs-In(i, b) do
4             RemoveLeftPath(p → original, i − 1)
5             for q ∈ Arcs-Out(i − 1, p → origin) do
6                 if (p == q) then
7                     RemoveArc(q, Arcs-Out(i − 1, p → origin)
8             RemoveArc(p, Arcs-In(i, b))
```

3.3.1 Adding New Constraints During Search

As explained earlier, the algorithm uses Gauss-Jordan Elimination with modulo p and makes it simpler to solve the constraints. We can add more constraints during the search. In the following, we present how we can add a new constraint to previous examples.

According to example 3.2.1 after using Gauss-Jordan Elimination with modulo 5, we reach a more straightforward format which is presented below. Now we add a new constraint to the previous format.

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 4 & 1 \end{array}\right] \quad \text{Add new constraint} \quad 2x_1 + 3x_2 + 3x_3 \equiv 2 \ mod\ 5$$

The algorithm does not start from scratch: instead, it adds the new constraint to the current straightforward format then again uses Gauss-Jordan Elimination with mod p.

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 4 & 1 \\ 2 & 3 & 3 & 2 \end{array}\right] -2R_1 + R_3 \quad \sim \quad \left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 4 & 1 \\ 0 & 3 & 3 & 3 \end{array}\right] -3R_2 + R_3$$

$$\sim \left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right] -4R_3 + R_2 \quad \sim \quad \left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right]$$

Here the solution is $(x_1, x_2, x_3) = (2, 1, 0)$. All the variables are non-parametric.

For Example 3.2.2, the simpler format is

$$\left[\begin{array}{cccc|c} 1 & 0 & 0 & 2 & 1 \\ 0 & 1 & 4 & 4 & 4 \end{array}\right] \text{ Add new constraint: } 2x_1 + 3x_2 + 4x_3 + 2x_4 \equiv 1 \ mod\ 5$$

$$\left[\begin{array}{cccc|c} 1 & 0 & 0 & 2 & 1 \\ 0 & 1 & 4 & 4 & 4 \\ 2 & 3 & 4 & 2 & 1 \end{array}\right] -2R_1 + R_3 \quad \sim \quad \left[\begin{array}{cccc|c} 1 & 0 & 0 & 2 & 1 \\ 0 & 1 & 4 & 4 & 4 \\ 0 & 3 & 4 & 3 & 4 \end{array}\right] -3R_2 + R_3$$

$$\sim \left[\begin{array}{cccc|c} 1 & 0 & 0 & 2 & 1 \\ 0 & 1 & 4 & 4 & 4 \\ 0 & 0 & 2 & 1 & 2 \end{array}\right] 3R_3 \quad \sim \quad \left[\begin{array}{cccc|c} 1 & 0 & 0 & 2 & 1 \\ 0 & 1 & 4 & 4 & 4 \\ 0 & 0 & 1 & 3 & 1 \end{array}\right] -4R_3 + R_2$$

$$\sim \left[\begin{array}{cccc|c} 1 & 0 & 0 & 2 & 1 \\ 0 & 1 & 0 & 2 & 0 \\ 0 & 0 & 1 & 3 & 1 \end{array}\right]$$

Here we already have one parametric variable. The graphs are shown below.

For the first constraint;

$$x_1 + 0x_3 + 2x_4 \equiv 1 \ mod\ 5$$

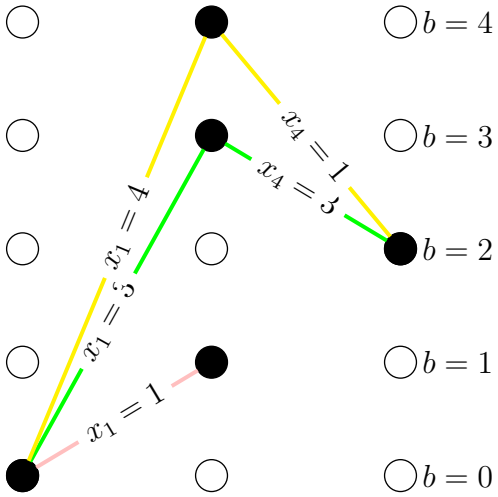

Figure 3.12 graphical representation for the first constraint of Example 3.2.2 after adding the new constraint

The value 1 is filtered out of the domain of x_1, and value 3 is filtered out of the domain of x_4. For the second constraint;

$$x_2 + 0x_3 + 2x_4 \equiv 0 \ mod \ 5$$

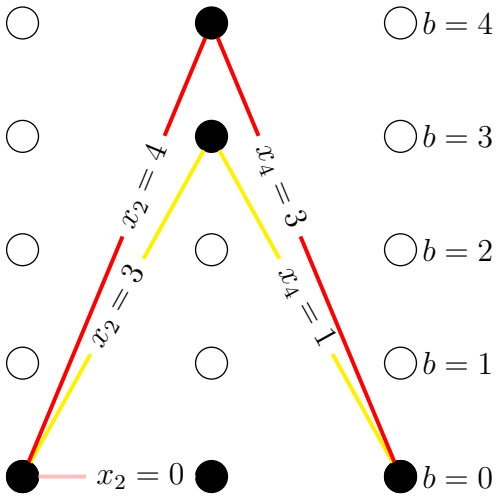

Figure 3.13 graphical representation for the second constraint of Example 3.2.2 after adding the new constraint

The value 0 is filtered out of the domain of x_2, and value 4 is filtered out of the domain of

x_4. For the third constraint;

$$x_3 + 3x_4 \equiv 1 \ mod \ 5$$

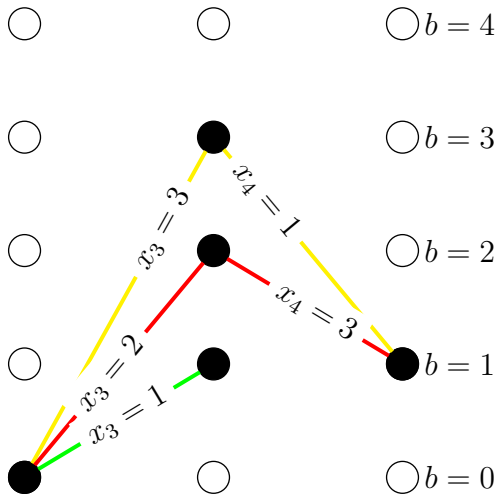

Figure 3.14 graphical representation for the third constraint of Example 3.2.2 after adding the new constraint

The value one is filtered out of the domain of x_3, and value four is filtered out of the domain of x_4.

3.4 Using Linear Equalities in Modular Arithmetic for Approximate Model Counting

3.4.1 Algorithm

Now we present the hashing-based approximate model counting. We have n variables, each of domain m and modular p which must be a prime number equal or greater than the domain size of variables. As we explain in Section 2.2.5, the size of the cells that are neither too large nor too small and the expected number of elements in each cell is $\frac{|R_f|}{p}$. So the choice of p is important.

According to the number of variables and number of chosen constraints the algorithm randomly generates constraints. The Gauss-Jordan elimination algorithm with modular arithmetic is called to make the constraint(s) simpler. According to the value of p and the number of random constraints the solution space is divided into a smaller area. By generating the first constraint the solution space is divided into p cells. Then for the second generated con-

straint, one of the cells is divided into p cells and so on. The graph incrementally computes the number of solutions in one small cell.

3.5 Illustration Of the Approach

Counting the number of solutions of the problem could be a time-consuming task. By generating Modular p constraint which we call "ModPHash" function, we can divide the solution space into small cells.

According to Example 1.1.1 the number of solutions without any modPHash function is 29400. In this example the domain of variables is $[1, \ldots, 5]$ and p must be the smallest prime number greater than the domain size. If we add $p = 7$ to the model, the solution space is divided to 7 small cells. Then the algorithm counts the number of solutions in one cell. We expect the number of solutions for one modPHash will be approximately 4200.

$$\frac{29400}{7} \approxeq 4200$$

Table 3.1 represents the number of solutions in a small cell for the different right-hand sides. The first column in the table presents the right-hand side of one given random constraint. Column 2 shows the number of solutions in one cell. The third column shows the coefficients of random constraint. According to the table the approximate count for the number of solutions in each cell is near to what we expect.

Table 3.1 The results of adding a modPHash constraint with different u

right-hand side	number of solutions	coefficients of random constraint
u=0	4196	[1 6 3 5 3 1 3 0 2 6]
u=1	4200	[1 4 3 4 0 0 4 3 6 5]
u=2	4202	[4 2 1 5 5 0 6 1 5 2]
u=3	4203	[3 1 3 3 4 5 6 1 0 2]
u=4	4204	[2 1 3 3 2 3 0 2 0 4]
u=5	4196	[2 1 6 6 3 1 2 0 0 2]
u=6	4200	[3 3 2 1 2 6 2 2 4 6]

3.6 Study of Counting Solutions of Two Constraints with Shared Variables

In this section we work on two constraints with shared variables and try to count the number of solutions, but that is not part of the final implementation.

3.6.1 One Shared Variable

In this part we consider two constraints A and B with modulo p. Let n_A be the number of solutions of constraint A and n_B be the number of solutions of constraint B that both are computed by f and g to find the exact number of solutions of the combination directly.

The number of solutions of $A \wedge B$ with no shared variable is equal to $n_A \times n_B$. We consider x_1 as shared variable for both constraints A and B. Let n_A^v be the number of solutions of constraint A where $x_1 = v$, and n_B^v be the number of solutions of constraint B where $x_1 = v$

Now we state that the number of solutions of constraint $A \wedge B$ with shared variable x_1 is equal to:

$$\sum_{v \in D_{x_1}} n_A^v \times n_B^v \leq n_A \times n_B$$

To hold the equality we can write;

$$\frac{1}{2}n_A \times \frac{1}{2}n_B + \frac{1}{2}n_A \times \frac{1}{2}n_B = \frac{1}{2}(n_A \times n_B)$$

if $n_A^0 = \frac{1}{2}n_A$, and $n_A^1 = \frac{1}{2}n_A$ for constraint A, and the same for constraint B.

Now consider;

$$\begin{cases} n_A^v = n_A & v = 0 \\ n_A^v = 0 & \text{Otherwise} \end{cases}$$

Moreover the same for constraint B. The above inequality will be equal.

The following example shows two constraints A and B with modulo 7. The domain of variables is $D = \{0, 1, 2, 3, 4\}$. The shared variable for these two constraints is x_3.

$$A: \quad 2x_1 + 4x_2 + x_3 + 3x_4 \equiv 3 \; mod \; 7$$

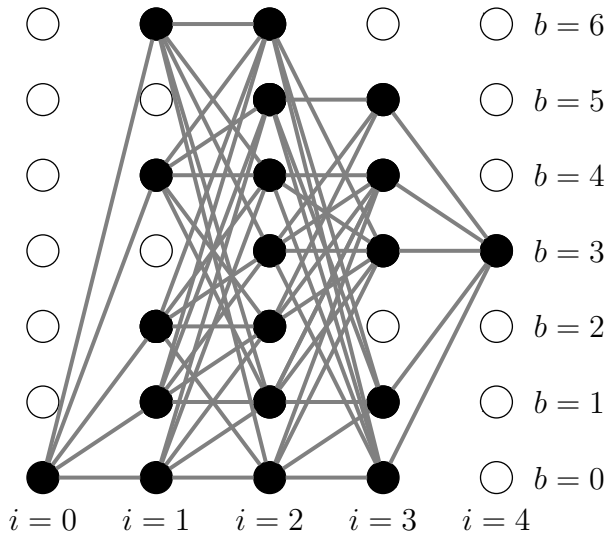

Figure 3.15 The graphical representation for constraint A

The number of solutions is equal to 89. Now we fix the value of variable $x_3 = 0$.

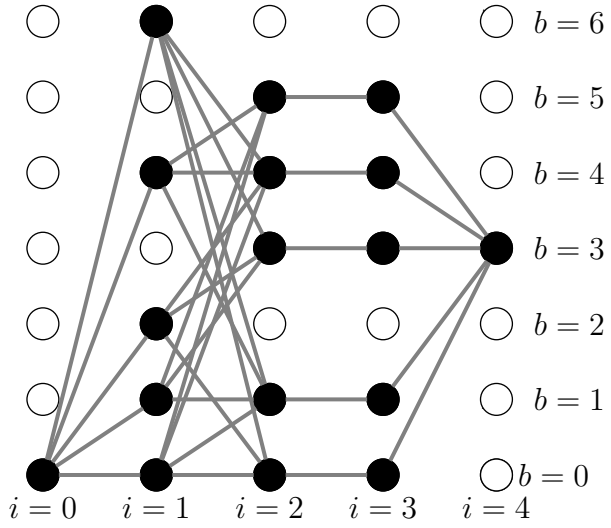

Figure 3.16 The graphical representation of constraint A when $x_3 = 0$

The number of solutions will be 17. We can easily find the other fixed value for variables x_3 in this example;

- If variable $x_3 = 0$ the number of solutions is equal to 17.

- If variable $x_3 = 1$ the number of solutions is equal to 18.
- If variable $x_3 = 2$ the number of solutions is equal to 18.
- If variable $x_3 = 3$ the number of solutions is equal to 18.
- If variable $x_3 = 4$ the number of solutions is equal to 18.

We can sum all the solutions for each value to find all the solutions for constraint A.

$$\sum_{x_3 \in D} n_A = 89$$

$$B: \quad 3x_5 + x_6 + 3x_3 + 4x_7 \equiv 3 \; mod \; 7$$

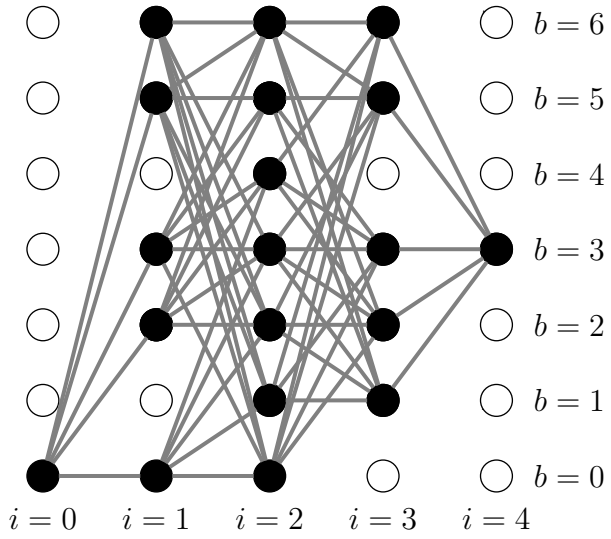

Figure 3.17 The graphical representation for constraint B

The number of solutions is equal to 91. Now we fix value of variable $x_3 = 0$.

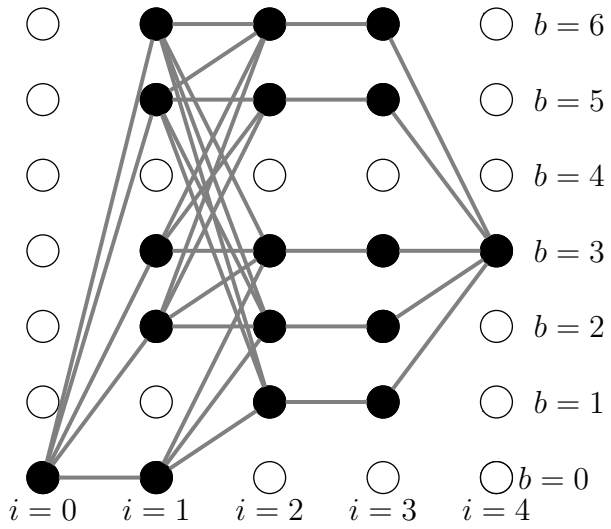

Figure 3.18 The graphical representation of constraint B when $x_3 = 0$

The number of solutions for variable $x_3 = 0$ is equal to 18.

- If variable $x_3 = 0$ the number of solutions is equal to 18.
- If variable $x_3 = 1$ the number of solutions is equal to 19.
- If variable $x_3 = 2$ the number of solutions is equal to 19.
- If variable $x_3 = 3$ the number of solutions is equal to 18.
- If variable $x_3 = 4$ the number of solutions is equal to 17.

For two constraints:

$$\begin{cases} 2x_1 + 4x_2 + x_3 + 3x_4 \equiv 3 \ mod \ 7 \\ 3x_5 + x_6 + 3x_3 + 4x_7 \equiv 3 \ mod \ 7 \end{cases}$$

- The number of solutions, for $x_3 = 0$ is equal to 17×18.
- The number of solutions, for $x_3 = 1$ is equal to 18×19.
- The number of solutions, for $x_3 = 2$ is equal to 18×19.
- The number of solutions, for $x_3 = 3$ is equal to 18×18.
- The number of solutions, for $x_3 = 4$ is equal to 18×17.

$$\sum_{x_3 \in D} n_A \wedge n_B = 1620$$

This number is the exact number of solutions of two constraints A and B with one shared variable.

3.6.2 Two Shared Variables

The next example shows 2 constraints with 2 shared variables x_2 and x_3. In this specific example we fixed variables x_2 and x_3 to value 1. The domain for the rest of the variables is $D = \{0, 1, 2, 3, 4\}$.

$$A: \quad 2x_1 + 4x_2 + x_3 + 3x_4 \equiv 3 \ mod \ 7$$

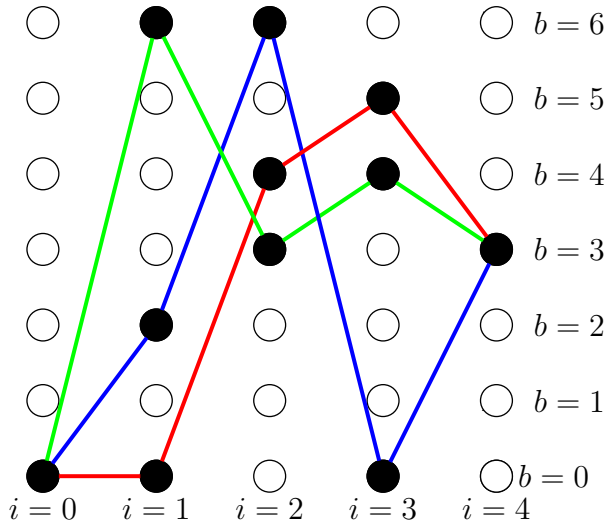

Figure 3.19 graphical representation of constraint A with two shared variables

The three paths are specified by different colors.Thus the three feasible solutions are (x_1, x_2, x_3, x_4) = $(0, 1, 1, 4)$, $(1, 1, 1, 1)$ and $(3, 1, 1, 2)$.

$$B: \quad 3x_5 + x_2 + x_3 + 4x_6 \equiv 3 \ mod \ 7$$

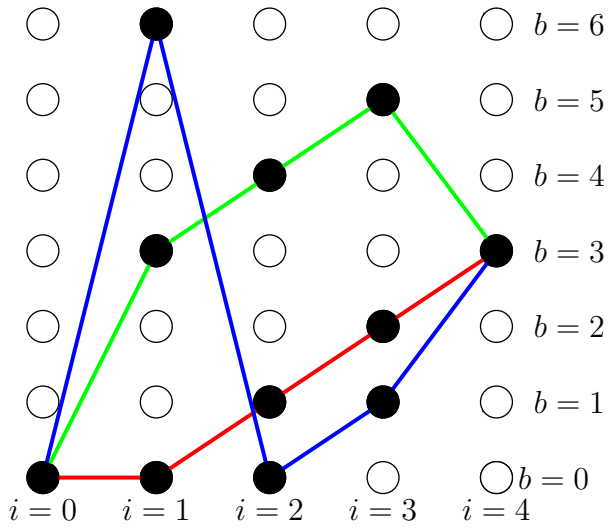

Figure 3.20 graphical representation of constraint B with two shared variables

The three paths are specified by different colors. Thus the three feasible solutions are $(x_5, x_2, x_3, x_6) = (0, 1, 1, 2)$, $(1, 1, 1, 3)$ and $(2, 1, 1, 4)$.

For two constraints:

$$\begin{cases} A: \quad 2x_1 + 4x_2 + x_3 + 3x_4 \equiv 3 \ mod \ 7 \\ B: \quad 3x_5 + x_2 + x_3 + 4x_6 \equiv 3 \ mod \ 7 \end{cases}$$

By multiplying the number of solutions of given constraints we can find the number of solutions in general. All feasible solutions are $(x_1, x_2, x_3, x_4, x_5, x_6)$= $(3, 1, 1, 2, 0, 2)$, $(3, 1, 1, 2, 1, 3)$, $(3, 1, 1, 2, 2, 4)$, $(0, 1, 1, 4, 0, 2)$, $(0, 1, 1, 4, 2, 4)$, $(1, 1, 1, 1, 1, 3)$, $(0, 1, 1, 4, 1, 3)$, $(1, 1, 1, 1, 0, 2)$ and $(1, 1, 1, 1, 2, 4)$.

3.6.3 Challenges in Counting the Number of Solutions in Inner Layers

If the shared variables do not appear in consecutive layers of both graphs, we can no longer rely on the computed f and g values to derive the exact number of solutions of the combination directly. We consider the possible values of the first and last layers are specified, and we want to count the number of paths between these two layers. In the following example there are four variables which have different domains. The domain for variable x_1 is $\{0, 1, 2, 3, 4\}$, x_2 is $\{3\}$, x_3 is $\{2, 3, 4\}$, and x_4 is $\{1\}$.

$$A: \quad 2x_1 + 4x_2 + x_3 + 3x_4 \equiv 3 \ mod \ 7$$

The following graph shows the possible values that x_1 and x_4 can take, and we want to know how many paths exist between these two layers (feasible solutions for x_2 and x_3).

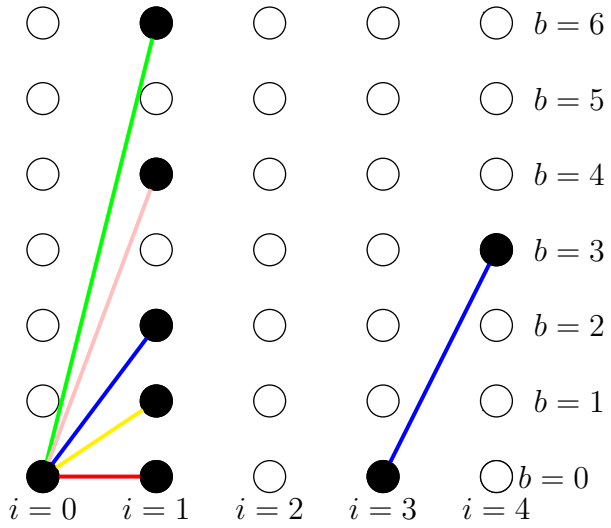

Figure 3.21 graphical representation of possible paths between variables x_1 and x_2

The domain for variable x_2 is $\{3\}$ and the coefficient for x_2 is 4, the domain for variable x_3 is $\{2, 3, 4\}$ and the coefficient for x_3 is 1. So, we can write $(3 \times 4) + 2 \bmod 7 = 0$, $(3 \times 4) + 3 \bmod 7 = 1$ and $(3 \times 4) + 4 \bmod 7 = 2$ in $f(i, b)$ if we sum these numbers with b we can find which nodes have paths to the node(s) in the last layer.

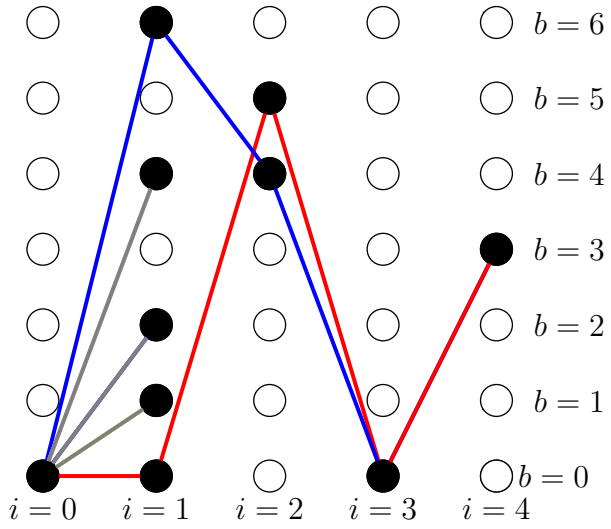

Figure 3.22 graphical representation of existing paths between two variables x_1 and x_2

It shows, we have two paths from f(1,0) and f(1,6) to f(3,0).

The following graph shows the feasible solution of the example if we count inner layers in a graph. The two red and blue paths are the solutions of this example.

In general, we can state that $f(i_{\text{first layer}}, b)+[(\text{coefficient}(x_{\text{first layer}+1})\times\text{domain}(x_{\text{first layer}+1}))\bmod p+ \ldots + (\text{coefficient}(i_{\text{last layer}-1}) \times \text{domain}(x_{\text{last layer}-1}) \bmod p)] = f(i_{\text{last layer}}, b)$

We may choose not to solve a problem precisely because the complexity is too high. So we prefer to consider approximate counting and find the upper bound instead of the exact number of solutions.

CHAPTER 4 EXPERIMENTS

In this chapter we report on experiments set up to answer the following research questions:

1. Does our approach reduce the computation time of model counting in CP compared to a straightforward enumeration of the solutions?

2. How accurate is our approximate approach to model counting?

In Figure 4.1 we present the classes diagram for our CP model. In the "model" class we define our model such as variables and constraints. Table constraint is defined in this class. Also we add universal hashing constraints inside the model class. In the "modPSystem" class, we define a Gauss-Jordan Elimination function to simplify the constraint system. Moreover, an addHash function gives us this ability to add a new constraint during the search. In "modPHash" we build the graph and use a daemon function for the incremental version of our model.

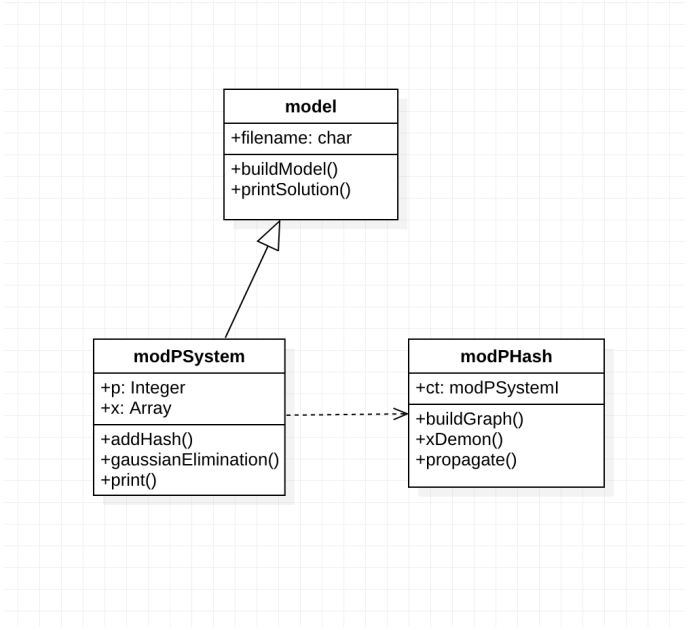

Figure 4.1 Classes diagram for CP model

.

4.1 Experimental Set up

All the examples in this chapter were implemented in C++11 within IBM ILOG CP version 1.6.

As a CP model, we use a single table constraint with forbidden tuples which by changing the number of variables, domain size, and the number of forbidden tuples allows us better control over the experiments. We use function IloTableConstraint to build the model based on tuples.

Algorithm 7: Table constraint

```
1 TupleSet forbiddenValues(n);
2 for each i in nbForbiddenValues do
3 |   tuple[i-1] = array[n];
4 |   for each j in n do
5 |   |   tuple[i-1][j] = random value in [1,..., m];
6 |   forbiddenValues.add(tuple[i-1]);
7 model.add(TableConstraint( n, m, forbiddenValues));
```

The formula to compute the exact number of solutions for each instance is $m^n - f$. The tuples are generated for 2 different instance sizes $n = 5$ and $n = 10$. For the smaller instances ($n = 5$) we choose 10, 20, and 30 as domain size and the number of forbidden tuples was chosen as 1% or 10% of the total number of solutions. For the larger ones ($n = 10$) we choose the prime number 5, 7, and 11 for the value of domain size, and 0.1% or 0.01% of the total number of solutions for the number of forbidden tuples.

4.2 Results

For all our experiments, we report average results over 10 runs.

Table 4.1 shows the computation time to enumerate (and thus count) all the solutions with 0 to 4 added mod p hashing constraints in non-incremental version. Here each time a graph is created from scratch. As we can see it is really slow except for 4 mod p that there are very few solutions to enumerate.

To show the computational advantage of not recomputing from scratch each time, we use the incremental version. The following results in Table 4.2 are close to what we expect. In this table we show the number of results and computation time for our CP model with varying number of modP constraints. We use a time limit of 120 hours for all the instances. In table "-" denotes timed out.

Table 4.1 The computation time average over 10 runs to count the solutions of our CP model with a varying number of modP constraints (non incremental version)

(n,m,f)	0 modP Time (s)	1 modP Time (s)	2 modP Time (s)	3 modP Time (s)	4 modP Time (s)
(5,10,1000)	0.49	4.70	7.86	2.65	0.30
(5,10,10000)	0.95	1.50	0.39	0.47	0.14
(5,20,32000)	43.36	1001.28	1536.46	279.05	8.15
(5,20,320000)	596.27	855.09	1002.69	499.79	23.38
(5,30,24300)	293.55	16692.26	55119.00	2519.76	154.37
(5,30,243000)	3930.03	5424.18	18615.50	3632.16	225.13

Table 4.2 The average results over 10 runs of counting the solutions of our CP model with a varying number of modP constraints (incremental version)

benchmark	0 modP		1 modP		2 modP		3 modP		4 modP	
(n,m,f)	#Solns	Time (s)	#Solns	Time (s)	#Solns	Time (s)	#Solns	Time (s)	#Solns	Time (s)
(5,10,1000)	99003.0	0.49	9002 .0	0.28	822.5	0.10	74.5	0.12	7.0	0.01
(5,10,10000)	90477.0	0.95	8227 .0	0.48	754.0	0.18	69.0	0.10	7.0	0.01
(5,20,32000)	3168146.0	43.36	137740.0	20.68	5987.6	6.29	260.6	5.24	11.3	0.04
(5,20,320000)	2895404.0	596.27	125918.5	274.46	5465.5	66.41	242.6	49.11	11.1	0.23
(5,30,24300)	24275716.0	293.55	783087.9	148.95	25258.2	47.46	814.9	38.61	26.2	0.11
(5,30,243000)	24058235.0	3930.03	776105.9	1987.92	25042.7	313.49	806.5	263.42	26.0	0.53
(10,5,9766)	9755868.0	207.52	1393707.0	99.02	199106.2	53.02	28436.5	32.72	4069.4	29.70
(10,5,97656)	9668441.0	3055.63	1381153.0	1430.09	197365.3	645.15	28202.5	353.61	4022.5	302.78
(10,7,28248)	282447001.0	12650.30	25676944.1	5384.96	2334234.0	2220.19	212226.3	1397.08	19303.8	1197.16
(10,7,282475)	282192905.0	284106.00	25653942.0	121664.00	2332316.0	49714.83	211986.1	23172.89	19274.1	31002.68
(10,9,34868)	3486749533.0	-	316976882.0	49313.96	28816114.0	21893.22	2619862.8	11607.40	238352.8	13583.77
(10,9,348678)	3486435723.0	-	316948503.0	426811.00	28813682.0	423900.00	2619512.0	391918.00	238209.0	376514.00

From the number of solutions, for k added modPHash, we approximate the original number of solutions by multiplying by p^k.

The accuracy of the results can be calculated as follows:

$$\%\text{Error} = \frac{|\#\text{Solns}_{\text{exact}} - \#\text{Solns}_{\text{approx}}|}{\#\text{Solns}_{\text{exact}}} \times 100$$

Table 4.3 reports the percentage of accuracy for the instances. The columns show the %Error for the varying number of modP constraints. As we can see for some instances the %Error is higher for the instance with a greater number of forbidden tuples compared to the same instances with fewer forbidden tuples. For example, the %Error in the second row for 4 modP hash compared to the first row is much higher. The reason is that by increasing the number of forbidden tuples the number of solutions in a small cell is decreased, and there is not an appropriate number of solutions in one random small cell. We know that the number of solutions in each cell is important. So the %Error for instance $(5, 10, 10000)$ is greater

than that of $(5, 10, 1000)$.

Table 4.3 Quality of approximation for different number of modPHash (average over 10 runs)

benchmark (n,m,f)	%Error: 1 modP	%Error: 2 modP	%Error: 3 modP	%Error: 4 modP
(5,10,1000)	0.0190000	0.5200000	0.1500000	3.5100000
(5,10,10000)	0.0220000	0.8300000	1.5000000	13.2700000
(5,20,32000)	0.0039000	0.0220000	0.0810000	0.1800000
(5,20,320000)	0.0240000	0.1430000	1.9400000	7.2800000
(5,30,24300)	0.0000300	0.0100000	0.0039000	0.2500000
(5,30,243000)	0.0043000	0.0320000	0.1320000	0.1940000
(10,5,9766)	0.0008300	0.0034000	0.0220000	0.1500000
(10,5,97656)	0.0038000	0.0250000	0.0510000	0.1070000
(10,7,28248)	0.0002100	0.0016000	0.0092000	0.0630000
(10,7,282475)	0.0002000	0.0061000	0.0130000	0.0001200
(10,9,34868)	0.0001000	0.0000074	0.0082000	0.0085000
(10,9,348678)	0.0000620	0.0005600	0.0038000	0.0330000

Table 4.4 reports the standard deviation over 10 runs. The goal of this table is to show how the results vary from the mean. As we can see in the table the distribution of data is approximately normal. In general for one modP constraint, the data points are spread out over a wider range of values. By adding more ModP constraints the results are close to what we expect.

4.3 Analysis of the Results

We started this chapter with two important questions which are the main goal of this research. According to the results which are presented we can see by adding modPHash we have a reduction in computation time. Moreover, we showed the accuracy of each result. As an example, for instance $(10, 5, 9766)$, computation time is 207.52 and the number of solutions is 9755868 without any modPHash. By adding 4 modPHash, we have a significant reduction in computation time. As Table 4.2 shows, the computation time is 29.70 and the average number of solutions is 4069.4. If we multiply 4069.4 by 7^4 (here p = 7), we reach 9770629.4 which is close to the exact number of solutions. For the larger instances, the %error is decreased.

If we compare the non-incremental version with an incremental version we can see that, there is a reduction in computation time in new results. Also, by adding more modP constraint the computation time is reduced.

Table 4.4 Quality of approximation for different number of modPHash (standard deviation over 10 runs)

benchmark (n,m,f)	%Error: 1 modP	%Error: 2 modP	%Error: 3 modP	%Error: 4 modP
(5,10,1000)	0.000708	0.001243	0.004931	0.000000
(5,10,10000)	0.000520	0.005340	0.021956	0.000000
(5,20,32000)	0.503450	0.001301	0.008994	0.054813
(5,20,320000)	0.000426	0.012331	0.011046	0.105808
(5,30,24300)	1.755030	0.000136	0.000528	0.006511
(5,30,243000)	4.927246	0.000319	0.002802	0.017897
(10,5,9766)	2.256357	0.000160	0.001561	0.002704
(10,5,97656)	7.793421	0.000510	0.001660	0.005326
(10,7,28248)	2.179091	2.488790	0.000372	0.002025
(10,7,282475)	8.772020	3.958581	0.007781	0.002451
(10,9,34868)	3.637108	2.790476	0.005600	0.006300
(10,9,348678)	8.347083	3.825592	0.002460	0.018560

Figure 4.2 indicates the error with respect to computation time for different modPHash for $n = 5$ instances. As we can see, without any modPHash, computation time is high but the error is zero. By adding ModPHash we decrease the computation time.

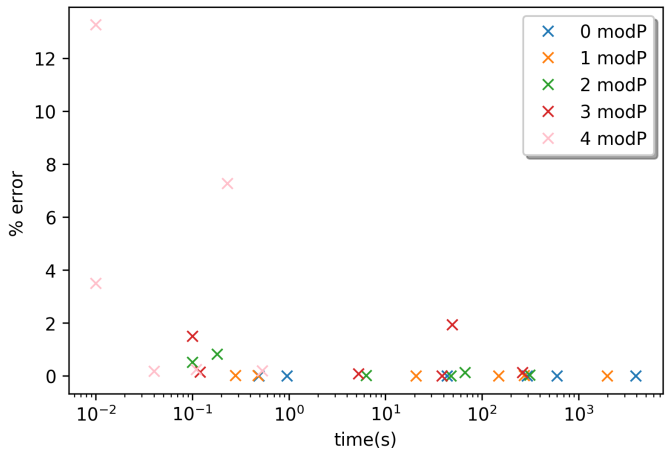

Figure 4.2 The %error for different modP for $n = 5$

.

Figure 4.3 shows the error for different modPHash for $n = 10$ instances.

As we can see by increasing the size of the domain the computation time increases but we have a reduction in error. Also, we compare the results for different f. As the results show for a larger number of forbidden tuples, the error and computation time increase. It suggests

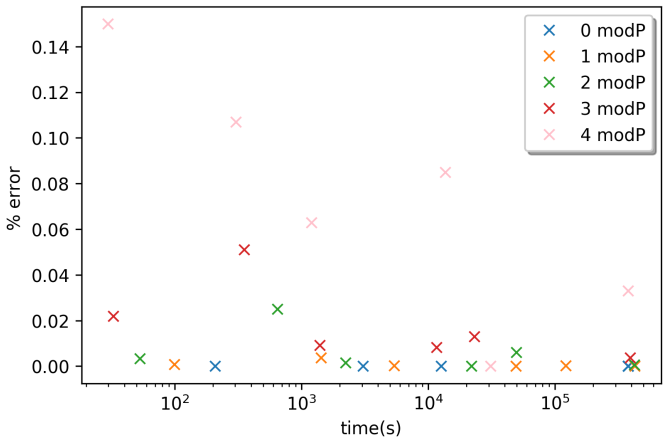

Figure 4.3 The %error for different modP for $n = 10$

.

that for a larger solution space the algorithm works better.

The figures show three added modPHash gives the best result. The percentage of error is low and we have a good reduction in computation time.

CHAPTER 5 CONCLUSION

In this thesis, we worked on counting the solutions of CSPs by using modular arithmetic. This final chapter presents a brief conclusion of the work and future research in this area.

5.1 Summary of Work

Model counting and sampling are important issues in AI. This dissertation worked on linear equality constraints in modular arithmetic. All the previous works in this area worked on counting and sampling for problems with binary domains, here we introduce modular p constraints generalized to non-binary domains. We represent a layered graph for each constraint and discuss how the graph counts the number of solutions and how it filters. We use Gauss-Jordan Elimination with modular arithmetic to simplify the constraints system. Modular arithmetic plays an essential role in Gauss-Jordan Elimination algorithm. We never perform division, and all the coefficients remain in F_p. This is a big advantage because we know Gauss-Jordan Elimination cannot be used on integer variables unless we are in modular arithmetic. The main work of this research is providing incremental filtering algorithm for the system of equations. We show the computation time for non-incremental version to show the computational advantage of not recomputing from scratch each time. The incremental manner helps us never start from scratch when the domains of variables change or when we want to add new constraints during the search. Another effort is counting solutions of two constraints with shared variables. Moreover, counting the number of solutions in inner layers in a layered graph.

The algorithm uses the hashing-based technique and divides the solutions space into p cells, then randomly chooses one of the cells and counts the number of solutions. As we expected, this algorithm approximately computes the number of solutions with a reduction in computation time. Also, we can claim the approximate counts of the algorithm are accurate.

5.2 Limitations

Our model for instances with fewer variables and very many solutions works better because for instances whose number of variables is huge need to build a huge graph and needs more time. Moreover, the presented model is useful for limited applications.

5.3 Future Research

This was a first step towards a CP model counter. This work has shown the feasibility and promise of the approach on non-binary domains in CP. We will need to implement an algorithm adding one modP constraint at a time until stopping criterion. Also we haven't used yet the ability to count the number of solutions of individual mod p constraints. During this research, new questions came up. One of the ideas for future research could find the best size of each cell in a solution space. Moreover, we will present the tolerance and confidence level in theory.

REFERENCES

[1] K. S. M. Supratik Chakraborty and M. Y. Vard, "Discrete sampling and integration in high dimensional spaces." Tutorial at Conference on Uncertainty in Artificial Intelligence, New York, 2016.

[2] L. G. Valiant, "The complexity of enumeration and reliability problems," *SIAM Journal on Computing*, vol. 8, no. 3, pp. 410–421, 1979.

[3] D. Roth, "On the hardness of approximate reasoning," *Artificial Intelligence*, vol. 82, no. 1-2, pp. 273–302, 1996.

[4] H. Kautz and B. Selman, "Ten challenges redux: Recent progress in propositional reasoning and search," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2003, pp. 1–18.

[5] T. Sang, P. Beame, and H. A. Kautz, "Performing bayesian inference by weighted model counting," in *AAAI*, vol. 5, 2005, pp. 475–481.

[6] J. D. Park, "Map complexity results and approximation methods," in *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 2002, pp. 388–396.

[7] C. Domshlak and J. Hoffmann, "Probabilistic planning via heuristic forward search and weighted model counting," *Journal of Artificial Intelligence Research*, vol. 30, pp. 565–620, 2007.

[8] L. E. Sucar, "Probabilistic graphical models," *Advances in Computer Vision and Pattern Recognition. London: Springer London. doi*, vol. 10, pp. 978–1, 2015.

[9] G. Pesant, "A constraint programming primer," *EURO Journal on Computational Optimization*, vol. 2, no. 3, pp. 89–97, 2014.

[10] ——, "Achieving domain consistency and counting solutions for dispersion constraints," *INFORMS Journal on Computing*, vol. 27, no. 4, pp. 690–703, 2015.

[11] C. Lecoutre and R. Szymanek, "Generalized arc consistency for positive table constraints," in *International conference on principles and practice of constraint programming*. Springer, 2006, pp. 284–298.

[12] K. Mehlhorn and S. Thiel, "Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2000, pp. 306–319.

[13] J.-C. Régin, "A filtering algorithm for constraints of difference in csps," in *AAAI*, vol. 94, 1994, pp. 362–367.

[14] M. A. Trick, "A dynamic programming approach for consistency and propagation for knapsack constraints," *Annals of Operations Research*, vol. 118, no. 1, pp. 73–84, 2003.

[15] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013, pp. 200–216.

[16] F. B. S. D. T. Pitassi, "Dpll with caching a new algorithm for# sat and bayesian inference," 2002.

[17] W. Wei and B. Selman, "A new approach to model counting," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2005, pp. 324–339.

[18] W. Wei, J. Erenrich, and B. Selman, "Towards efficient sampling: Exploiting random walk strategies," in *AAAI*, vol. 4, 2004, pp. 670–676.

[19] K. S. Meel *et al.*, "Constrained sampling and counting: Universal hashing meets sat solving." in *AAAI Workshop: Beyond NP*, 2016.

[20] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Balancing scalability and uniformity in sat witness generator," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[21] A. Ivrii *et al.*, "On computing minimal independent support and its applications to sampling and counting," *Constraints*, vol. 21, no. 1, pp. 41–58, 2016.

[22] C. P. Gomes, A. Sabharwal, and B. Selman, "Near-uniform sampling of combinatorial spaces using xor constraints," in *Advances In Neural Information Processing Systems*, 2007, pp. 481–488.

[23] S. Ermon *et al.*, "Embed and project: Discrete sampling with universal hashing," in *Advances in Neural Information Processing Systems*, 2013, pp. 2085–2093.

[24] ——, "Taming the curse of dimensionality: Discrete integration by hashing and optimization," in *International Conference on Machine Learning*, 2013, pp. 334–342.

[25] S. Chakraborty *et al.*, "Approximate probabilistic inference via word-level counting." in *AAAI*, vol. 16, 2016, pp. 3218–3224.